RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHRSTUHL
INFORMATIK XI
SOFTWARE FÜR
EINGEBETTETE
SYSTEME

Diplomarbeit

Anforderungen an einen Model-Checker für Matlab/Simulink

Requirements for a model checker for Matlab/Simulink

Jacob Palczynski
Matrikelnummer: 209337

5. Dezember 2005

Gutachter: Prof. Dr.-Ing. Stefan Kowalewski
Prof. Dr. rer. nat. Wolfgang Thomas

Sicherheitskritische Systeme werden zunehmend modellbasiert entworfen. Eine Plattform für modellbasierten Entwurf ist Simulink von The MathWorks, Inc. Um automatisch zu verifizieren, dass ein System gewisse Anforderungen erfüllt, wird Model-Checking eingesetzt.

In der vorliegenden Diplomarbeit werden Anforderungen an einen Model-Checker für Matlab/Simulink formuliert. Um zu evaluieren, ob ein Model-Checker diese Anforderungen erfüllt, wird eine Suite von Simulink-Modellen entwickelt. Die Erproung dieser Evaluierungssuite geschieht anhand zweier kommerzieller Model-Checker, OSC EmbeddedValidator und TNI Valiosys Safety-Checker Blockset. Die Evaluierungsergebnisse werden präsentiert und bewertet, und Schlüsse für die Weiterentwiclung der Suite werden gezogen.

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht.

Aachen, den 5. Dezember 2005

(Jacob Palczynski)

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1. Einleitung

Eingebettete Systeme und die zugehörige Software werden immer öfter modellbasiert entworfen. Regler und Strecke werden graphisch beschrieben, der gewünschte Code für den Regler automatisch erzeugt. Damit wird die fehlerträchtige manuelle Umsetzung des Designs in Code beim konventionellen Entwicklungsprozess vermieden. Der modellbasierte Entwurf erleichtert die Zusammenarbeit von Entwicklern aus verschiedenen Wissensgebieten. Gerade bei der Entwicklung eingebetteter Systeme, die an der Grenze zwischen Regelungstechnik und Informatik geschieht, ist das von Vorteil.

Eine weit verbreitete Plattform für modellbasierten Entwurf ist Simulink von The MathWorks, Inc. Es handelt sich dabei um eine auf Matlab aufbauende Umgebung, in der konfigurierbare Blöcke verschiedene Funktionen zur Verfügung stellen und mit Daten- und Signalleitungen verbunden werden. Stateflow erweitert Simulink um endliche Transitionssysteme. Die entworfenen Simulink-Modelle können wir in der Entwurfsumgebung simulieren und damit auch Testfälle erstellen und überprüfen.

Eingebettete Software findet in sicherheitskritischen Bereichen immer weitere Verbreitung. Beispiele hierfür sind Produkte der Luft- und Raumfahrt- und der Automobilindustrie, aber auch Produktionsanlagen der chemischen Industrie. In diesen Bereichen sind finanzielle Belastungen die geringsten Folgen von Softwarefehlfunktionen. Es drohen Gefahren für Mensch und Umwelt.

Aus diesen Gründen muss sichergestellt sein, dass ein eingebettetes System korrekt funktioniert. Tests reichen hierfür nicht aus, da nicht alle Möglichkeiten des Systemverhaltens abgedeckt werden können. Um zu kontrollieren, ob ein System innerhalb einer Spezifikation arbeitet, greifen wir zu einem Mittel der formalen Verifikation, dem Model-Checking. Beim Model-Checking lassen wir automatisch überprüfen, ob ein Transitionssystem eine bestimmte Spezifikation erfüllt. Existiert ein Verhalten des Systems, dass die Spezifikation nicht erfüllt, schlägt die Verifikation fehl.

Innerhalb des modellbasierten Entwurfs können wir die Verifikation mittels Model-Checking in verschieden Phasen nutzen. Während der Modellierung überprüfen wir, ob Teilsysteme eines größeren Entwurfs die sie betreffenden Spezifikationen erfüllen. Fehler können wir somit früh erkennen und korrigieren, später aufwendige Änderungen werden vermieden. Ist das Gesamtsystem vollständig modelliert, kontrollieren wir, ob dieses alle Anforderungen erfüllt. Diese beiden Ansätze für das Model-Checking setzen auf dem graphisch erstellten Modell auf. Ein anderer Ansatz für das Model-Checking ist die Verifikation des aus dem Modell gewonnenen Codes.

1.1. Über diese Diplomarbeit

Der Entwurf von sicherheitskritischen Systemen mit Matlab/Simulink und Stateflow wirft die Frage auf, welche Möglichkeiten wir haben, die entworfenen Systeme zu verifizieren.

1.1.1. Aufgabenstellung

Um diese Frage grundlegend zu beantworten, müssen wir erst ermitteln, welche funktionale und nichtfunktionale Anforderungen ein Model-Checker für Simulink im Hinblick auf den Einsatz beim modellbasierten Entwurf erfüllen muss. Eine zentrale Aufgabe dieser Diplomarbeit ist es, diese Anforderungen zu formulieren und zu ermitteln, wie sich diese schnell und objektiv überprüfen lassen.

1.1.2. Struktur der Diplomarbeit

In Kapitel 2 geben wir einen Überblick über Model-Checking im Allgemeinen und über Model-Checking von Simulink-Modellen im Besonderen. Grundlegende Informationen zu Matlab, Simulink und Stateflow stammen vor allem aus den offiziellen Dokumentationen von The MathWorks [Mat05b], [Mat05a], [Mat05c] und [Mat05d]. Weitere Quellen sind [ABRW04] und [Tiw02]. Wir stellen die Model-Checker OSC EmbeddedValidator und TNI Safety-Checker Blockset im gleichen Kapitel vor. Dazu stützen wir uns auf die Dokumentation der Hersteller [OSC05], [TNI05].

Im Rahmen dieser Diplomarbeit fragen wir, ob wir diese Werkzeuge ohne Einschränkungen im Entwicklungsprozess einsetzen können. Wir betrachten dabei insbesondere folgende Aspekte:

1. Welche syntaktischen Konstrukte akzeptiert der Model-Checker?
2. Wie verhält sich seine Laufzeit mit steigenden Beweisgrößen?

Auf Basis dieser Fragen formulieren wir in Kapitel 3 die Anforderungen, die ein Model-Checker für Simulink erfüllen soll.

Die formulierten Anforderungen sollen überprüft werden. Aus diesem Grund entwerfen und implementieren wir in Kapitel 4 eine Evaluierungssuite. Diese besteht neben Simulink-Modellen aus einem Vorgehensmodell, dass den Evaluierungsprozess beschreibt.

Wir zeigen, dass die Suite sich eignet, Model-Checker auf die Anforderungen hin zu evaluieren, in dem wir sie anhand zweier Model-Checker erproben. Geleitet durch das Vorgehensmodell evaluieren wir OSC EmbeddedValidator und TNI Safety-Checker Blockset. Den Verlauf und die Ergebnisse der Evaluierung präsentieren wir in Kapitel 5.

Basierend auf den Ergebnissen der Erprobung ziehen abschließend Schlussfolgerungen für die weitere Entwicklung der Evaluierungssuite und für den Produktiveinsatz aktueller Model-Checker. Ferner gehen wir in Kapitel 6 auf die Notwendigkeit eines neuen Model-Checkers für Simulink ein.

Der Anhang ab Seite 90 enthält vertiefendes Material.

1.1.3. Eingesetzte Software

Als Basis setzen wir Matlab Release 14 SP2 (Matlab 7.0.4, Simulink 6.2 und Stateflow 6.2). Als Betriebssystem dient Microsoft Windows XP Professional. Mit Model-Checkern OSC EmbeddedValidator in der Version 2.0 und TNI Safety-Checker Blockset in der Version 2.2 erproben wir die Suite. EmbeddedValidator 2.0 benötigt zur Codeerzeugung dSPACE TargetLink 2.0.7, für die Darstellung der Benutzeroberfläche benötigt es einen X11-Server. Cygwin wird in der Version 1.5.10 installiert, dessen X11-Server wir verwenden.

Diese Diplomarbeit wird in LaTeXgeschrieben, als Editoren verwenden wir Kile 1.7 unter Linux (KDE 3.4) und TeXnicCenter 1 Beta 6.13 unter Windows XP. Zusätzlich wurden für

die Suite und die Diplomarbeit Microsoft Office und Visio, OpenOffice und Adobe Acrobat benutzt.

1.1.4. Weiteres Material

Neben dem vorliegenden Dokument umfasst die Diplomarbeit die Evaluierungssuite mit einem Benutzerleitfaden, graphischen Darstellungen des Vorgehens bei der Evaluierung und an den Evaluierungsprozess angepassten Tabellen zur Ergebniserfassung. Dieses Material befindet sich auf der beiliegenden CD-ROM.

2. Grundlagen

Der erste Abschnitt dieses Kapitels beschäftigt sich mit den Grundlagen des Model-Checking. Die mathematischen Konstrukte, auf denen wir Model-Checking durchführen, werden eingeführt, sowie das Model-Checking-Problem selbst. Die Ausführungen basieren auf der Vorlesung „Model-Checking“ [TL03], gehalten von Prof. Wolfgang Thomas an der RWTH Aachen im Wintersemester 2003/2004.

Im weiteren Verlauf dieses Kapitels stellen wir zunächst Matlab, Simulink und Stateflow kurz vor und gehen bei Simulink und Stateflow auf die Simulationssemantik ein. Mit der Simulationssemantik bezeichnen wir die Art und Weise, in der Simulink die Modelle abarbeitet. Sie legt fest, wie sich die Blöcke verhalten, und in welcher Reihenfolge sie im Modell aktiviert werden. Wir präsentieren die Möglichkeiten, die sich bei der Modellierung und bei der Codeerzeugung ergeben.

Abschließend beschäftigen wir uns mit dem Model-Checking von Simulink-Modellen. Wir diskutieren die möglichen Ansätze und stellen zwei Model-Checker genauer vor.

2.1. Model-Checking

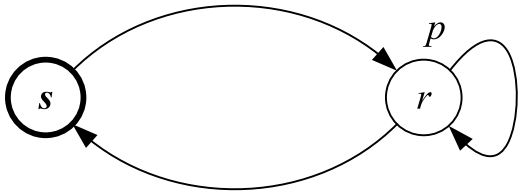
Model-Checking ist eine Methode, Programme automatisch zu verifizieren. Das Ziel der Verifikation ist der mathematische Beweis, dass ein Programm eine präzise Spezifikation erfüllt. Damit unterscheidet sich die Verifkation von den anderen Validierungsmethoden, Simulation und Test. Bei diesen Techniken können wir nicht beweisen, dass ein Programm eine Spezifikation erfüllt, sondern höchstens, dass es sie nicht erfüllt.

Für die Verifikation stehen zwei unterschiedliche Methoden zur Verfügung, Deduktion und Model-Checking. Letzteres hat den für uns entscheidenden Vorteil, automatisierbar zu sein.

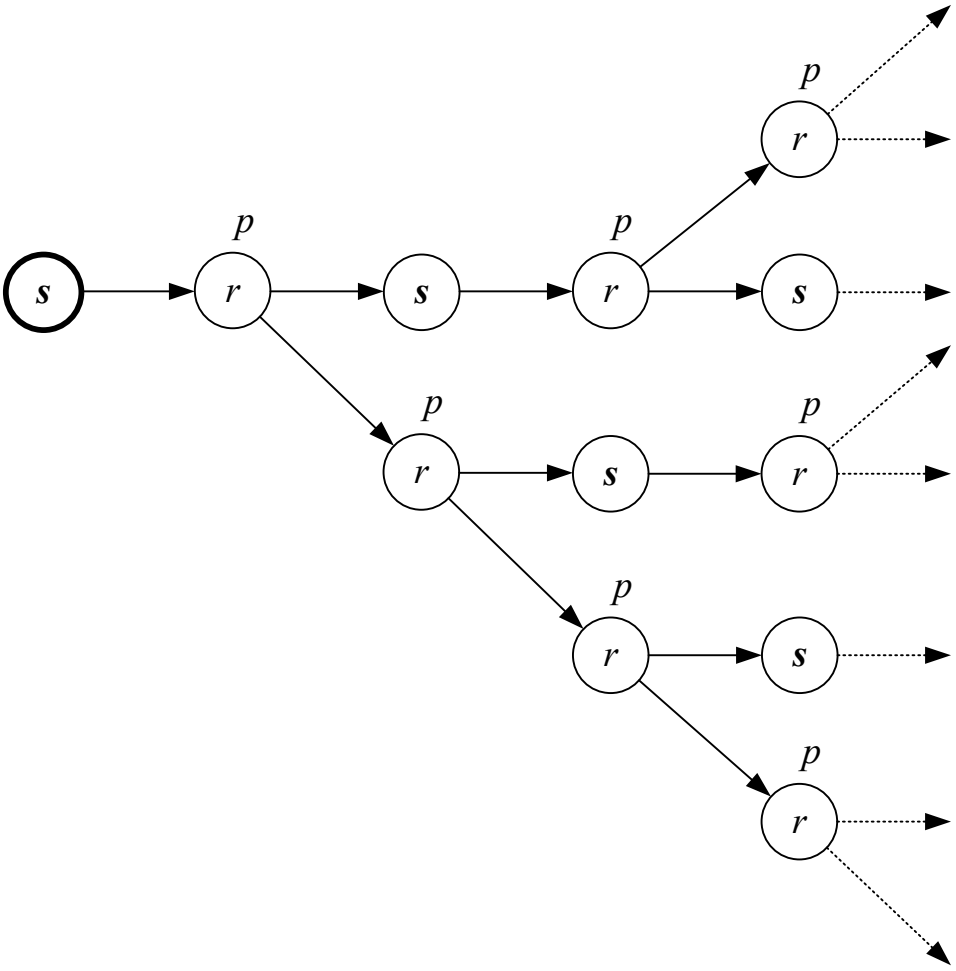
2.1.1. Model-Checking-Problem

Bevor wir das Model-Checking-Problem formulieren, benötigen wir eine formale Darstellung der Objekte, mit denen sich das Model-Checking beschäftigt.

Das Programm modellieren wir mit einer Kripke-Struktur. Dabei handelt es sich um einen gerichteten Graphen mit einem ausgezeichneten Anfangsknoten, dessen Knoten beschriftet sind (Abbildung 2.1(a), siehe auch Definition A.1.1). Die Beschriftung an einem Knoten repräsentiert die in diesem Knoten erfüllten Aussagen. Die Spezifikationen werden beim Model-Checking mittels Anforderung an unendliche Programmläufe formuliert. Wir benötigen daher eine Struktur, mit der wir die Läufe eines Programms beschreiben können. Dafür wickeln wir eine Kripke-Struktur $\mathfrak{K}$ zum Abwicklungsbaum $\mathfrak{K}_{\mathfrak{T}}$ ab. Hierbei handelt es sich um eine Kripke-Struktur, die in Form eines Baumes alle möglichen Pfade durch die ursprüngliche Struktur $\mathfrak{K}$ enthält (Abbildung 2.1(b), siehe auch Definition A.1.2). Mit dem Abwicklungsbaum einer Kripke-Struktur modellieren wir alle möglichen Programmabläufe des von der Kripke-Struktur repräsentierten Programms. Die Spezifikation für das Model Checking ist eine Eigenschaft E, die Aussagen über die Variablen $p_1, \ldots, p_n$ an den Baumknoten macht.



(a) Kripke-Struktur



(b) Abwicklungsbaum

Abbildung 2.1.: Beispiel einer Kripke-Struktur und ein Ausschnitt aus dem Abwicklungsbaum der Struktur

(a) Tempraloperatoren

Operator	Bedeutung
$\mathsf{X}p$	im nächsten Schritt gilt p
$p_1 \mathsf{U} p_2$	irgendwann gilt p_2, davor gilt immer p_1
$p_1 \mathsf{R} p_2$	immer gilt p_2, ausser p_1 gilt
$\mathsf{F}p$	irgendwann ab jetzt gilt p
$\mathsf{G}p$	ab jetzt gilt immer p

(b) Pfadquantoren

Pfadquantor	Bedeutung
$\mathsf{A}p$	auf allen ausgehenden Pfaden gilt p
$\mathsf{E}p$	es gibt einen ausgehenden Pfad, auf dem p gilt

Tabelle 2.1.: Nicht-formale Semantik der Temporaloperatoren (LTL und CTL) und Pfadquantoren (nur CTL); p, p_1 und p_2 sind jeweils Pfad-, bzw. Zustandsformeln.

Wir können jetzt das **Model-Checking-Problem** formulieren.

Gegeben:
- Kripke-Struktur $\mathfrak{K}$
- Eigenschaft E von Abwicklungsbäumen

Frage: Erfüllt der Abwicklungsbaum $\mathfrak{K}_\mathfrak{T}$ die Eigenschaft E?

Ausgabe: Entweder

- Bestätigung, dass $\mathfrak{K}_\mathfrak{T}$ E erfüllt oder
- Hinweis auf den Grund, warum $\mathfrak{K}_\mathfrak{T}$ E nicht erfüllt.

2.1.2. Temporallogiken

Mit der Eigenschaft E wollen wir Aussagen über die Zustände eines Programms und ihre Veränderung im Verlauf der möglichen Programmabläufe machen. Solche Aussagen werden mittels Temporallogiken formuliert. Beim Model-Checking sind die Logiken LTL (*linear time logic*) und CTL (*computation tree logic*) gebräuchlich. Dabei handelt es sich um Erweiterungen der Aussagenlogik. Beide Temporallogiken verfolgen einen unterschiedlichen Ansatz und sind nicht äquivalent.

LTL lässt Aussagen über Pfade in einer Struktur zu. Dafür stehen uns die Temporaloperatoren X, U, R, G und F zur Verfügung. LTL-Formeln sind induktiv aus aussagenlogischen Formeln, Temporaloparatoren und booleschen Operatoren zusammengesetzt. Tabelle 2.1(a) enthält eine nicht-formale Formulierung der Semantik der Temporaloperatoren. Eine LTL-Formel ist genau dann in einer Struktur erfüllt, wenn sie auf allen Pfaden vom Anfangspunkt aus erfüllt ist.

Beispiel 2.1.1. Folgende LTL-Formeln sind in der Kripke-Struktur in Abbildung 2.1 erfüllt:

- $\neg p \mathsf{U} p$
- $\mathsf{F}p$
- $\mathsf{XG}p \vee \mathsf{G}\,(\neg p \rightarrow \mathsf{X}p)$

Nicht erfüllt ist dagegen die Formel $\mathsf{XG}p$, da es einen Pfad gibt, auf dem immer wieder $\neg p$ gilt.

CTL ermöglicht Aussagen über Eigenschaften des Baumes, insbesondere über Verzweigungen in ihm. Dafür stehen zusätzlich zu den Temporalquantoren die Pfadquantoren A und E zur Verfügung (Tabelle 2.1(b)). In CTL darf ein Pfadquantor und ein Temporaloperator nur gemeinsam kombiniert in Paaren genutzt werden, wie beispielweise in $\mathsf{AG}p$ oder $\mathsf{E}\,(p_1 \mathsf{U} p_2)$.

Beispiel 2.1.2. Die CTL-Formel $\mathsf{EXEG}p$ ist in der Kripkestruktur in Abbildung 2.1 erfüllt. Die CTL-Formel $\mathsf{AG}\,(p \rightarrow \mathsf{AX}\neg p)$ ist nicht erfüllt.

Formale Definitionen zu LTL und CTL befinden sich in Anhang A.1. Model-Checking-Algorithmen und weitere Informationen zum Model-Checking bietet das Standardwerk von Clarke [CGP99].

2.1.3. Jenseits diskreter Systeme

Die oben vorgestellten Kripke-Strukturen sind diskret. Solche Strukturen reichen aber oft nicht aus, um reale Systeme zu beschreiben. Deswegen werden sie zu Zeitautomaten und in einem weiteren Schritt zu hybriden Systemen erweitert.

Bei Zeitautomaten [AD94] sind kontinuierliche Variablen definiert, deren Wert in Zuständen des Automaten mit einer konstanten Rate wächst; man nennt diese Variablen Uhren. Transitionen können abhängig von Uhrenvariablenwerten genommen werden. Wird eine Transition genommen, besteht die Möglichkeit, Uhrenvariablen auf den Startwert zurückzusetzen. Tatsächlich ist es mit Zeitautomaten möglich, zeitliche Zusammenhänge zu modellieren.

Während das Model-Checking-Problem auf diskreten, endlich-darstellbaren Systemen und Zeitautomaten entscheidbar ist, gilt das für hybride Systeme [Hen96] nicht mehr. Zustände hybrider Systeme bestehen aus diskreten Orten und einer Menge kontinuierlicher Variablen. Das Verhalten der kontinuierlichen Variablen wird in den Orten mit Differentialgleichungen beschrieben. Mit hybriden Systemen lassen sich reale physikalische Prozesse mit Schaltvorgängen kombinieren.

2.1.4. Schwierigkeiten beim Einsatz von Model-Checking

Beim Einsatz von Model-Checking auf reale Programme können diverse Schwierigkeiten auftreten. Eine davon ist, aus dem Programm eine Struktur zu gewinnen, die sich mit annehmbaren Zeit- und Ressourcenaufwand model-checken lässt. Ist dieses bei einfachen Programmen mit wenigen booleschen Variablen noch einfach, wächst mit dem Wertebereich der Variablen der Zustandsraum exponentiell an. Zudem können bei der Übersetzung des Programms in eine Kripke-Struktur Fehler auftreten, so dass man Spezifikationen für ein falsches System verifiziert.

Eine weitere Schwierigkeit betrifft die Spezifikation. Anforderungen an Softwaresysteme werden zunächst in natürlicher Sprache formuliert. Bei der Übersetzung in eine Spezifikationssprache eines Model-Checkers können Ungenauigkeiten und Fehler auftreten.

In welchem Umfang die oben genannten Schwierigkeiten den Benutzer betreffen, hängt von der Konstruktion des Model-Checkers ab. Bei aktuellen Model-Checkern müssen vor allem die Spezifikationen vom Benutzer in eine bestimmte Eingabesprache des Model-Checkers übersetzt werden. Das Ideal ist ein Model-Checker für eine Programmiersprache, dem ein beliebiges Programm eingegeben werden kann und der natürlichsprachige Spezifikationen akzeptiert.

2.2. Matlab – Simulink – Stateflow

Matlab, Simulink und Stateflow werden von The MathWorks entwickelt und vertrieben. Sie bauen aufeinander auf und bilden zusammen eine Entwicklungs- und Simulationsumgebung für wissenschaftlich-technische Probleme.

Matlab ist ein modular aufgebautes Softwarepaket für numerische Berechnungen. Der Name leitet sich von MATrix LABoratory ab und verdeutlicht, dass der Schwerpunkt auf Matrix- und Vektorrechnung liegt. Das Paket umfasst die Programmiersprache Matlab sowie eine Entwicklungsumgebung. Das Basismodul stellt uns neben Funktionen zur Programmablaufsteuerung und zur Ein- und Ausgabe umfangreiche mathematische Funktionen zur Verfügung. Die mathematischen Fähigkeiten von Matlab umfassen unter anderem Arithmetik, lineare Algebra, Fourieranalyse und numerische Integration. Ergebnisse und Daten können wir auf verschiedene Art und Weise zwei- oder dreidimensional darstellen und darüberhinaus graphische Benutzeroberflächen implementieren. Schließlich bietet das Basismodul Schnittstellen zu anderen Programmiersprachen und zu Hardwarekomponenten an [ABRW04].

Die Funktionalität des Basismoduls lässt sich durch Toolboxen erweitern. Eine besondere Toolbox stellt Simulink dar. Mit Simulink können wir Systeme, die sich zeitabhängig verhalten, graphisch entwerfen und anschließend simulieren. Modelliert werden Blockdiagramme, indem einzelne Funktionsblöcke mit Signalleitungen verbunden werden (Abbildung 2.2(b)). Die Blöcke sind in modularen Blockbibliotheken organisiert und umfassen in der Basisversion die in Abbildung 2.3(a) gezeigten Kategorien. Alle Blöcke sind in der Referenz [Mat05a] dokumentiert. Die einzelnen Kategorien enthalten die entsprechenden Blöcke. Als Beispiel ist in Abbildung 2.3(b) die Kategorie Math Operations gezeigt.

Einen besonderer Block ist der Stateflow-Block. Mit diesem fügen wir ereignisgesteuerte Systeme ein, die wir in einer Art Zustandsdiagramm modellieren (Abbildung 2.2(c)).

2.2.1. Syntaktische Aspekte

Im folgenden betrachten wir grundlegende syntaktische Aspekte von Simulink und Stateflow. Simulink-Modelle werden in speziellen Dateien (Endung `.mdl`) abgespeichert. Die Dateien haben ein spezielles Format, in dem Informationen zu den verwendeten Blöcken, den Verbindungen zwischen ihnen und allgemeinen Modell-Einstellungen festgehalten werden. Die Struktur der Dateien und ein Teil der abgespeicherten Parameter sind in [Mat05a] dokumentiert.

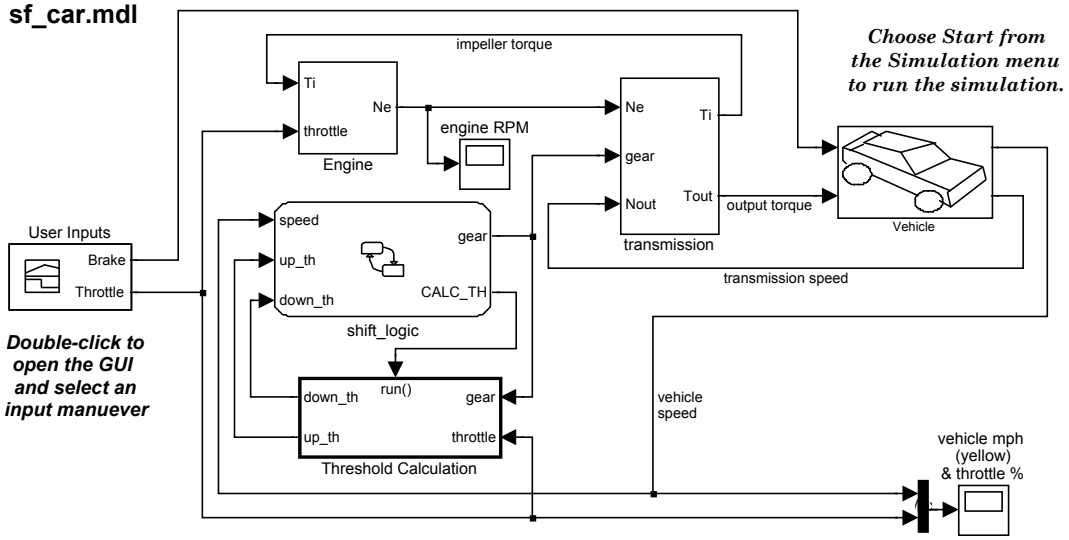
Bemerkung 2.2.1. Die `.mdl`-Dateien sind mit einem üblichen Editor lesbar. Mit Werkzeugen wie SimEx von IT Power Consultants[1] können sie geparst und in andere Formate (bei SimEx ist es XML) übersetzt werden.

2.2.1.1. Simulink

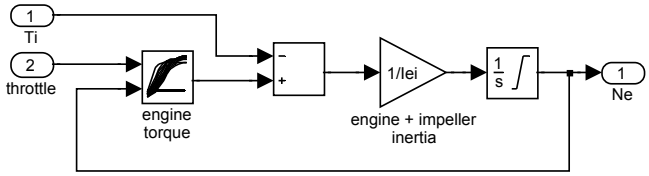
Blöcke Fügen wir einen Block in ein Modell ein, können wir ihn durch einen blockspezifischen Maskendialog konfigurieren (Abbildung 2.4). Variieren wir die Werte der Parameter eines Blocks, erhalten wir unterschiedliche **Ausprägungen** dieses Blocks. Die Parameter sind vom Block abhängig und umfassen unter anderem:

- Datentyp des Ausgangs,

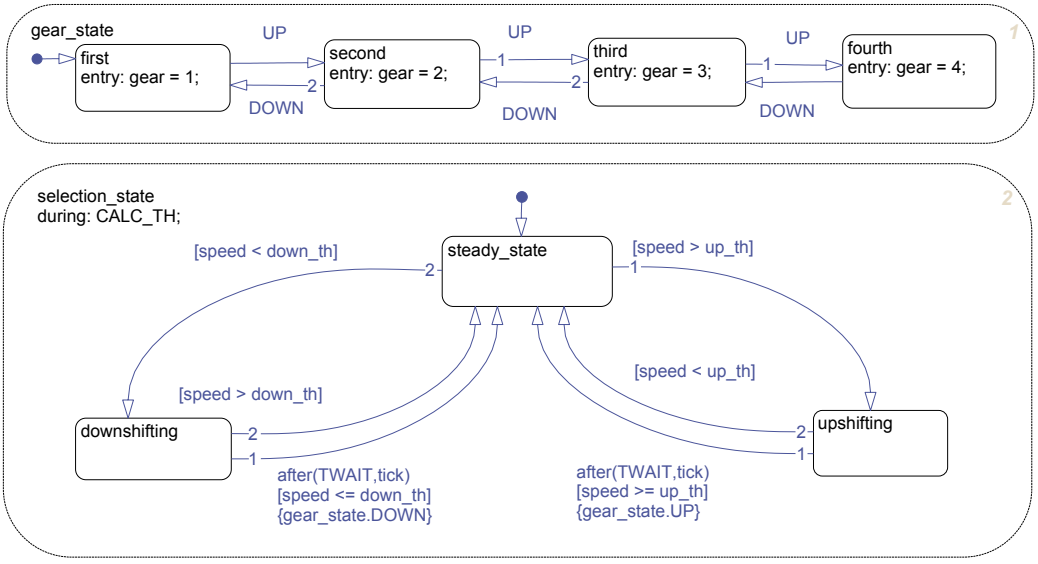
[1] `http://www.itpower.de/simex.html`



(a) `sf_car.mdl`

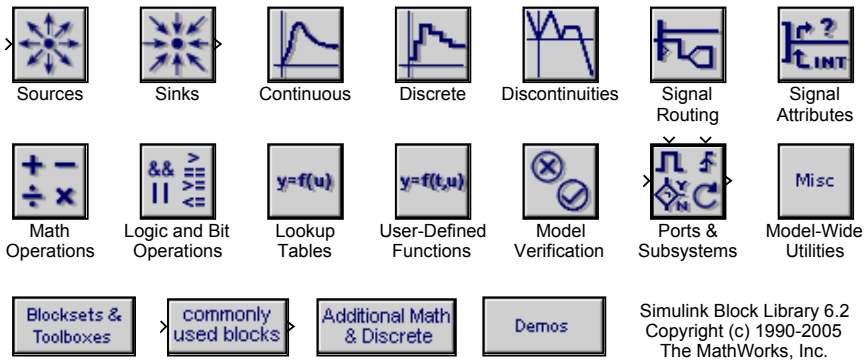


(b) `sf_car.mdl/Engine`

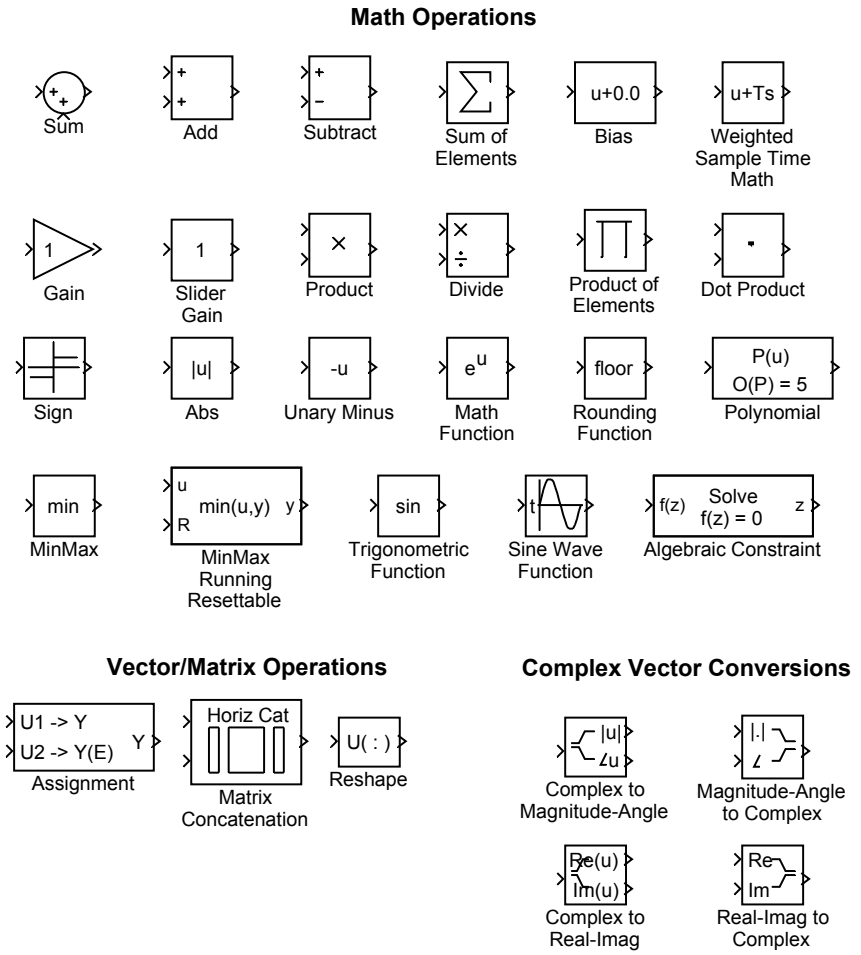


(c) sf_car.mdl/Shift Logic

Abbildung 2.2.: Beispiel für ein Simulink-Modell, `sf_car.mdl`. Engine (b) ist ein normales Subsystem, shift_logic (c) ein Stateflow-Chart, Threshold Calculation ein Function-call Subsystem in (a).



(a) Simulink-Bibliothek



(b) Math Operations

Abbildung 2.3.: Kategorien der Simulink-Blockbibliothek mit einem Beispiel, Math Operations

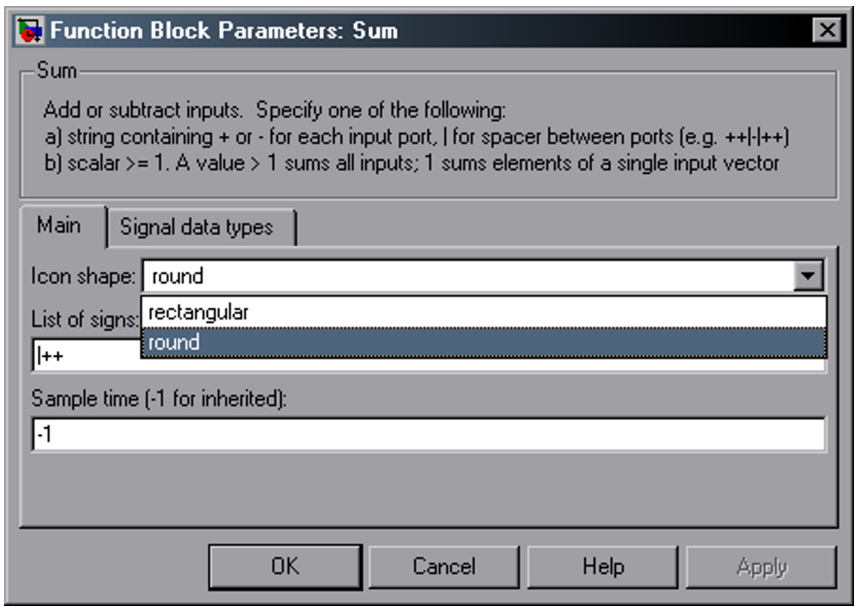


Abbildung 2.4.: Beispiel für einen Maskendialog: Sum-Block

- Anzahl der Eingänge,
- Belegung der Eingänge,
- Operator und
- Modus des Operators.

Diese Auflistung ist nicht vollständig, in der Simulink-Onlinehilfe und in [Mat05a] finden wir für jeden Block eine Übersicht über die jeweiligen Parameter.

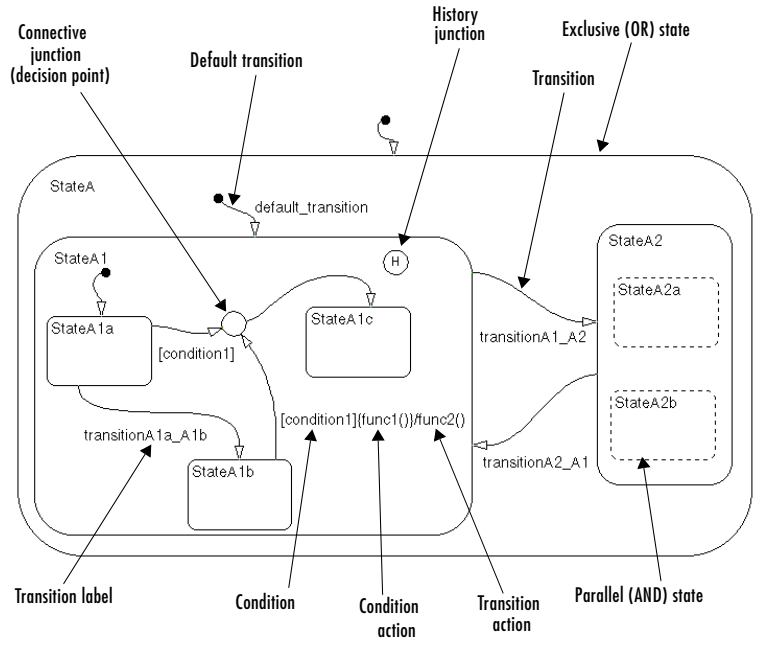
Große Modelle modularisieren wir, in dem wir mehrere Blöcke zu Subsystemen zusammenfassen. So gewinnen wir zum einen mehr Übersicht, zum anderen können wir diese Subsysteme wie Blöcke kopieren und in anderen Modellen verwenden. Um Subsysteme bedingt ausführen zu lassen, können sie mit zusätzlichen Trigger-, Funktionsaufruf- oder Freigabe-Eingängen (Triggered, Function-call oder Enabled Subsystem) versehen werden. Solche Subsysteme sind automatisch **atomar**, d.h. sie werden bei der Simulation als ein Block betrachtet (Abbildung 2.2(a)). Wollen wir selbst erstellte Subsysteme in mehreren Modellen nutzen, ist es sinnvoll diese in einer eigene Bibliothek zusammenzufassen. Zusätzlich können wir Subsysteme maskieren, indem wir einen Maskendialog definieren, über den wir auch Parameter an das Subsystem übergeben können.

Fügen wir Subsysteme aus einer eigenen Bibliothek in ein Modell ein, so wird nur eine Verknüpfung eingefügt. Damit werden Änderungen am Subsystem in der Bibliothek in das Subsystem im Modell propagiert. Änderungen im Subsystem im Modell können wir nur vornehmen, wenn wir die Verknüpfung in die Bibliothek auflösen. Genauere Informationen zu Arbeiten mit maskierten Subsystemen und Bibliotheken finden wir in [Mat05c].

Eine andere Art, eigene Funktionen für Simulink zu implementieren, sind S-Funktionen. Wir erstellen dabei einen eigenen Block, in dem wir die gewünschten Operationen in C, C++, Ada oder Fortran definieren. Weitere Informationen zu S-Funktionen erhalten wir in [Mat05d].

Länge [bit]	Ganzzahldatentyp mit Vorzeichen	Ganzzahldatentyp ohne Vorzeichen	Fließkomma-datentyp
8	int8	uint8, boolean	
16	int16	uint16	single
32	int32	uint32	double

Tabelle 2.2.: Datentypen in Simulink



Komponente	Übersetzung
(parallel) State	(paralleler) Zustand
Label	Beschriftung
Condition	Bedingung
(Default) Transition	(Standard-) Transitionssegment
Junction	Knoten(punkt)

Abbildung 2.5.: Komponenten eines Stateflow-Charts (aus [Mat05b])

Signale Daten zwischen den Blöcken werden über Signalleitungen ausgetauscht. Der Datentyp wird durch den ausgebenden Block bestimmt, bei den meisten können wir diesen über den Maskendialog frei wählen. Zusätzlich ist es uns möglich, bei manchen Blöcken den Eingangsdatentyp einzuschränken.

Uns stehen in der Grundversion von Simulink Ganzzahl und Fließkommadatentypen mit 8, 16 und 32 bit Länge zur Verfügung (Tabelle 2.2). Der Datentyp boolean stellt einen Sonderfall dar, da er nur die Werte 0 und 1 annimmt, aber intern durch einen Wert vom Typ uint8 realisiert wird. Signale können 0-, 1- oder 2-dimensional sein (Skalar, Vektor, Matrix), zusätzlich wird zwischen reellen und komplexen Signalen unterschieden (siehe auch [Mat05c]).

2.2.1.2. Stateflow

Jeder Stateflow-Block beinhaltet ein Chart. Die graphischen Komponenten eines Stateflow-Charts sehen wir in Abbildung 2.5. Wir werden neben den originalen englischen Begriffen auch die in der Tabelle dieser Abbildung aufgeführten Übersetzungen benutzen. Da in [Mat05b] für einen einzelnen Transitionspfeil der Begriff „transition segment“ benutzt wird, halten wir uns hier auch an diese Bezeichnung.

Es gibt zwei Arten von Zuständen: parallele oder AND-Zustände und exklusive oder OR-Zustände. Wie wir in Abbildung 2.5 sehen, können wir in einen Zustand weitere Zustände einfügen. Wir können mit Hilfe von Ober- und Unterzuständen Charts hierarchisch aufbauen. Jedem Zustand müssen wir einen in seinem Oberzustand eindeutigen Namen zuordnen. Außerdem beschriften wir ihn bei Bedarf mit Anweisungen, die zu bestimmten Zeitpunkten der Simulation durchgeführt werden (in Abbildung 2.5 nicht dargestellt). Die allgemeine Form einer Zustandbeschriftung lautet (aus [Mat05b]):

```
name/
entry:entry actions
during:during actions
exit:exit actions
bind:data_name, event_name
on event_name:on event_name actions
```

Dafür wird eine eigene Sprache, Action Language, benutzt.

Mit Transitionssegmenten verbinden wir Zustände und Knotenpunkte untereinander. Knoten dienen dabei sowohl als Verzweigungsstelle, falls sie mehrere ausgehende Transitionssegmente haben, als auch als Zusammenführungen, falls mehrere eingehende Transitionssegmente vorliegen. Auch die einzelnen Transitionssegmente können wir in Action Language beschriften, die allgemeine Form einer Transitionssegmentbeschriftung lautet (aus [Mat05b]):

```
event_trigger[condition]{condition_action}/transition_action
```

Mehr zur Syntax von Action Language finden wir in [Mat05b].

Ein besonderes Segment stellt das Standardtransitionssegment dar. Es startet weder in einem Knoten noch in einem Zustand, und jede Hierarchieebene mit exklusiven Zuständen oder Knoten muss ein solches Segment enthalten.

Zu den graphischen Komponenten kommen die nicht-graphischen Elemente: Daten und Ereignisse. Diese definieren wir

- auf Chart-, oder Zustandsebene mit lokalem Gültigkeitsbereich,
- als Schnittstellen zu Simulink (Ein-, Ausgänge, Trigger, ...) oder
- als Parameter, die in Matlab mit Werten belegt werden.

Lokale Daten und Ereignisse stehen dem Objekt, in dem sie definiert wurden, und dessen Unterobjekten zur Verfügung.

2.2.2. Semantik

Wir beschreiben, wie Simulink-Modelle bei der Simulation abgearbeitet werden. Damit werden die Semantiken von Simulink und Stateflow informell definiert. Ein Beispiel einer formellen Definition der Semantiken von Simulink und Stateflow finden wir in [Tiw02].

2.2.2.1. Simulink

Als erstes definieren wir die Semantik eines Simulink-Blockdiagramms, wir beziehen uns dabei auf die in [Mat05c] gemachten Angaben.

Definition 2.2.2 (Simulink-Blockdiagramm)**.** Bei Simulink-Blockdiagrammen handelt es sich **zeitbasierte Blockdiagramme**. Deren Semantik wird durch folgende Eigenschaften charakterisiert:

1. Zeitbasierte Blockdiagramme beschreiben Beziehung zwischen Signalen und Zustandsvariablen in Abhängigkeit von der Zeit. Die Lösung eines Blockdiagramms erhalten wir, indem wir diese Beziehungen über der Zeit auswerten, die zwischen den wählbaren Anfangs- und Endzeitpunkt liegt. Jede solche Auswertung bezeichnen wir als Zeit- oder Simulationsschritt.

2. Signale repräsentieren zeitveränderliche Werte, die für alle Zeitpunkte zwischen Anfangs- und Endzeitpunkt definiert sind.

3. Blöcke repräsentieren Gleichungsmengen, die die Beziehungen zwischen Signalen und Zustandsvariablen beschreiben. Die Gleichungsmenge eines Blocks beschreibt die Beziehung zwischen Eingangs- und Ausgangssignalen sowie den Zustandsvariablen dieses Blocks. Blockparameter dienen als Koeffizienten in den Gleichungen.

Die Auswertung der Gleichungsmenge eines Blockdiagramms an aufeinanderfolgenden Zeitschritten ergibt die **Simulation** des durch das Modell repräsentierten Systems.

Bei den Zustandsvariablen handelt es sich um zwischengespeicherte Werte der Berechnung des vorhergehenden Zeitschritts. Welche Werte zwischengespeichert werden, hängt vom Block ab. Je nach Block sind die Zustandsvariablen kontinuierlich oder diskret in der Zeit.

Jedem Block können wir eine Abtastrate vorgeben. Aus den Abtastraten aller Blöcke wird die Schrittweite für die Simulation bestimmt.

Bemerkung 2.2.3. Die Gleichungsmengen der einzelnen Blöcke sind nicht öffentlich dokumentiert. Ferner werden sie bei Bedarf bei Simulink-Versionswechseln von The MathWorks verändert. Die Semantik steht uns somit nicht direkt zur Verfügung.

Die Simulation eines Modells wird durch den Solver vereinfacht wie folgt durchgeführt:

1. Das Modell wird kompiliert, dabei werden unter anderem folgende Aufgaben durchgeführt:

   Hierarchie verflachen: Nicht-atomare Subsysteme werden durch die in ihnen enthaltenen Blöcke ersetzt.

   Ordnung auf Blöcken bestimmen: Zur Bestimmung der Ausgänge werden in Blöcken ohne Zustandsvariablen die Eingangswerte benötigt. Aus diesem Grund werden erst Blöcke mit Zuständen behandelt und Abarbeitungsreihenfolge der anderen Blöcke dem Signalfluss folgend geordnet.

   Abtastraten vererben: Blöcken ohne definierte Abtastraten bekommen diese von den Vorgängern im Signalfluss weitergegeben.

2. Beim Einbinden wird neben der Speicherreservierung auch aus der Blockordnung eine effiziente Abarbeitungsreihenfolge der Blockgleichungsmethoden erstellt.

3. Die Simulationsschleife wird zwischen Anfangs- und Endzeitpunkt der Simulation wiederholt:

a) Die Ausgänge werden berechnet. Die Reihenfolge der Blöcke wurde während der Einbindungsphase festgelegt.

b) Die Zustandsvariablen werden werden mit Hilfe des Solvers berechnet. Das Lösungsverfahren ist vom Typ der Zustandsvariable abhängig

c) Blöcke mit kontinuierlichen Zuständen werden auf Unstetigkeiten überprüft.

d) Der Zeitpunkt des nächsten Schritts wird berechnet.

Diese Auflistung erfasst nur einige Schritte, eine vollständige Übersicht finden wir in [Mat05c].

Befinden sich Blöcke in nicht-atomaren Subsystemen, werden diese bei der Simulation behandelt, als würden sie sich auf der Ebene des Subsystems befinden. Um zu erreichen, dass die in einem Subsystem enthaltenen Blöcke wie ein Block behandelt werden, müssen wir sie in ein atomares Subsystem einfügen.

Das in den Simulationsparametern des Modells festgelegte Lösungsverfahren des Solvers bestimmt hauptsächlich die Auswahl der Zeitpunkte, die in die Simulation des Modells einfließen. Die zur Verfügung stehenden Verfahren werden zum einen danach unterteilt, ob die Schrittweite fest oder variabel ist. Wird keine Schrittweite für die Simulation vorgegeben, dienen die Blockabtastraten als Basis für die Schrittweite. Beide Kategorien sind zusätzlich danach unterteilt, ob sie diskret oder kontinuierlich sind. Bei letzteren bestimmt der Solver die kontinuierlichen Zustände der Blöcke im aktullen Zeitschritt über numerische Integration. Als Berechnungsbasis dienen Zustände des vorhergehenden Zeitschritts und die Ableitungen dieser Zustände. Diskrete Lösungsverfahren berechnen lediglich den Zeitpunkt des nächsten Simulationsschritts, kontinuierliche Zustände werden nicht beachtet. Beide Arten der Lösungsverfahren gehen davon aus, dass diskrete Zustände von den Blöcken selbst aktualisiert werden.

Software für eingebettete Systeme läuft unter einem Echtzeit-Betriebssystem, die mit festen Zeitintervallen arbeiten. Entwerfen wir ein eingebettetes System mit Simulink, wählen wir sinnvollerweise ein Lösungsverfahren mit fester Schrittweite.

2.2.2.2. Stateflow

In [Mat05b] ist die Semantik von Stateflow-Charts ausführlich beschrieben (siehe auch Anhang A.2). Wir fassen zusammen, wie ein Stateflow-Chart in der Simulation behandelt wird.

Folgende axiomatische Eigenschaften bilden den Rahmen für das Verhalten von Stateflow.

1. *Ist ein Zustand aktiv, sind seine Oberzustände aktiv.*
2. *Ein Zustand oder Chart mit exklusiver Aufteilung hat maximal einen aktiven Unterzustand.*
3. *Ist ein paralleler Zustand aktiv, sind seine Nebenzustände mit höherer Priorität aktiv.*

Ferner gelten folgende Regeln, damit die Zustände konsistent sind:

1. *Ein aktiver OR-Zustand mit mindestens einem Unterzustand hat genau einen aktiven Unterzustand.*
2. *Alle Unterzustände eines aktiven parallelen Zustands sind aktiv.*
3. *Alle Unterzustände eines inaktiven Zustands sind inaktiv.*

Stateflow-Charts werden durch Ereignisse gesteuert. Diese Ereignisse haben zwei mögliche Quellen:

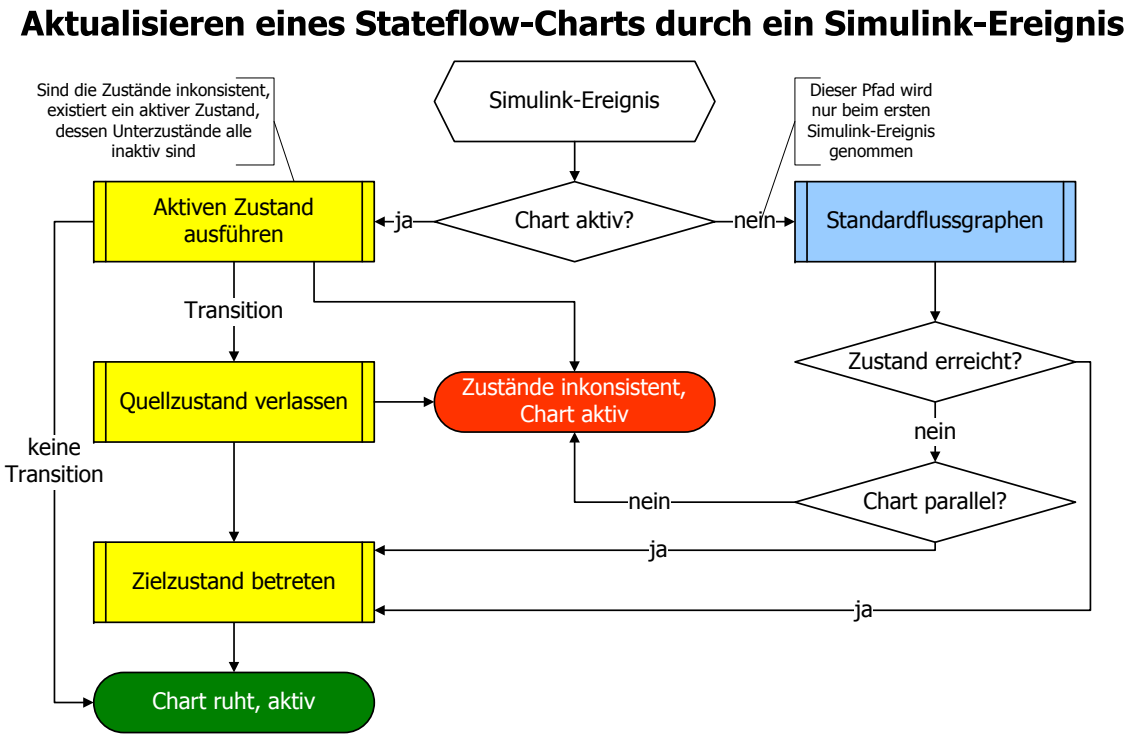


Abbildung 2.6.: Schritte beim Ausführen eines Stateflow-Charts, die Schritte auf Transitionen und Zuständen sind in Abbildung 2.7 aufgeschlüsselt.

1. Hat ein Chart Eingänge, werden von der Simulink-Ebene können direkt Ereignissignale eingegeben oder durch aktualisierte Eingangsdatensignale Ereignisse erzeugt. Hat ein Chart keine Eingänge, erzeugt der Solver bei jedem Zeitschritt ein Ereignis (zu den Schnittstellen zwischen Simulink und Stateflow siehe A.3).

2. Bei der Simulation werden im Chart implizite Ereignisse (Anhang A.3.1) erzeugt, die nur wirksam werden, wenn auf sie zugegriffen wird. Mit Action Language erzeugen wir bei Bedarf zusätzliche interne Ereignisse.

Tritt ein Ereignis auf, laufen die in Abbildung 2.6 gezeigten Schritte ab. Ein Chart wird in einem einzelnen Thread abgearbeitet. Damit das Chart konsistent bleibt, werden alle Aktionen, die durch ein Ereignis hervorgerufen werden (Abbildungen 2.6 und 2.7), atomar ausgeführt. Dadurch wird sichergestellt, dass zum Beispiel parallele Zustände einer Hierarchieebene in jedem Zeitschritt entweder alle gleichzeitig aktiv oder inaktiv sind. Aus der Tatsache, dass Stateflow in einem Thread abgearbeitet wird, ergibt sich die Notwendigkeit, auf parallelen Zuständen und ausgehenden Transitionssegmenten eine Ordnung zu definieren, nach der diese behandelt werden. Definieren wir keine eigenen Ordnungen, greift Stateflow auf die implizit erstellten zurück. Bei parallelen Zuständen ergibt sich diese direkt aus den Positionen der Zustände einer Hierarchieebene. Diese werden von oben nach unten und dann von links nach rechts geordnet.

Bei Transitionssegmenten ergibt sich die Ordnung aus mehreren Kriterien.

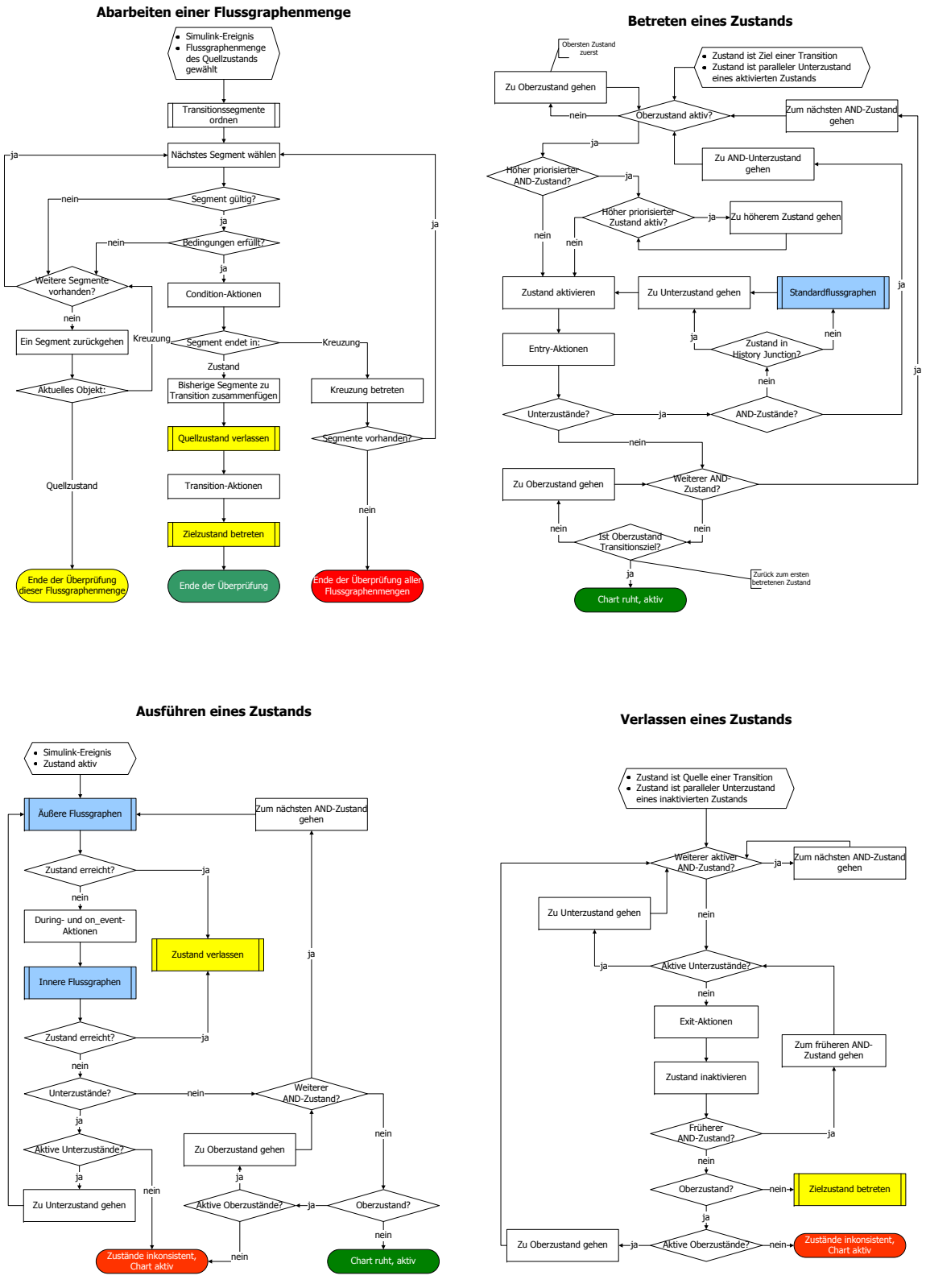


Abbildung 2.7.: Aufschlüsselung der Aktionen in Abbildung 2.6 (Erläuterung in Anhang A.2)

1. Je höher die Hierarchieebene des Ziels liegt, desto früher wird das Segment abgearbeitet.
2. Segmente mit gleicher Zielhierarchieebene werden nach ihrer Beschriftung geordnet:
   a) Beschriftung mit Ereignissen und Bedingungen
   b) Beschriftung mit Ereignissen
   c) Beschriftung mit Bedinungen
   d) Keine Beschriftung
3. Nach Zielhierarchieebene und Beschriftung äquivalente Segmente werden nach der geometrischen Position ihrer Startpunkte geordnet. Dabei werden sie aufsteigend im mathematisch positiven Sinn geordnet. Die Transition mit der niedrigsten Priorität liegt bei Zuständen direkt unterhalb der oberen linken Ecke, bei Knotenpunkten in der 12-Uhr-Position.

Transitionen werden aus den einzelnen Segmenten zusammengesetzt. Dafür werden die unmittelbaren und mittelbaren Transitionssegmente eines Zustands in drei Mengen von Flussgraphen eingeordnet.

- Alle Transitionssegmente, die Stardardtransitionssegmente enthalten und im gleichen Oberzustand starten, gehören zu Standardflussgraphen.
- Innere Flussgraphen umfassen alle Segmente, die in einem Zustand starten und diesen nicht verlassen.
- Äußere Flussgraphen enthalten alle Segmente, die den Quellzustand verlassen.

Die Flussgraphen enden in Zuständen. Die verschiedenen Mengen werden bei unterschiedlichen Schritten eines Zeitpunkts gebraucht (Abbildung 2.7) und sind nicht disjunkt.

Wie wir in Abbildung 2.6 sehen, wird mit dem ersten Ereignis die Standardflussmenge des Charts ausgeführt. Endet die genommene Transition in einem Zustand, wird dieser und seine Unterzustände betreten und als aktiv markiert. Bei den nächsten Ereignissen werden dann die Standardflussgraphen nicht mehr beachtet, sondern die inneren und äußeren Flussgraphen der aktiven Zustände. Endet die genommene Transition des Standardflussgraphen des Charts in einem Knoten, so werden in den nächsten Ereignissen wieder diese Standardflussgraphen ausgeführt.

Die erste Transition in der Ordnung innerhalb eines Flussgraphen, deren Bedingungen erfüllt sind, wird ausgeführt. Werden dabei Zustände betreten oder verlassen, werden die entsprechenden Entry- oder Exit-Aktionen ausgeführt. Kann keine Transition genommen werden, werden die During-Aktionen ausgeführt. Wegen des Zwangs, die erstmögliche Transition zu nehmen, können wir keine nichtdeterministischen Systeme mit Stateflow modellieren.

2.2.3. Codeerzeugung

Entwerfen wir mit Simulink ein eingebettetes System, können wir das Modell nicht direkt auf der Zielplattform nutzen. Aus dem Modell muss für das Ziel passender Code erzeugt werden, was mit dem Grundmodul von Simulink nicht möglich ist.

Von The MathWorks stehen drei separat zu erwerbende Toolboxen zur Auswahl.

- Stateflow Coder übersetzt Stateflow-Charts in C-Code.

- Real-Time Workshop erzeugt lauffähigen C-Code aus Simulink-Modellen.
- Real-Time Workshop Embedded Coder soll, besonders kompakten Code für eingebettete Systeme erzeugen.

Auch Dritthersteller liefern Werkzeuge zur Codeerzeugung aus Simulink-Modellen. Ein Beispiel dafür ist TargetLink von dSPACE. TargetLink ist ein umfangreiches Softwarepaket zur C-Codeerzeugung aus Matlab/Simulink. Modelle können wir entweder direkt mit TargetLink-eigenen Blöcken erstellen oder bestehende Simulink-Modelle automatisch in TargetLink übersetzen lassen. Aus TargetLink-Subsystemen erzeugen wir C-Code für verschiedene Zielplattformen. Wir gehen nicht genauer auf die Möglichkeiten von TargetLink ein, da wir das Werkzeug nur indirekt über EmbeddedValidator nutzen. Weitere Informationen bietet die Dokumentation von TargetLink [dSP04].

2.3. Model-Checking von Simulink-Modellen

Wir untersuchen die Möglichkeiten, die sich für das Model-Checking von Simulink-Modellen ergeben.

2.3.1. Ansätze für einen Model-Checker

Für das Model-Checking von Simulink Modellen gibt es zwei Ansatzpunkte, deren Vor- und Nachteile wir diskutieren.

2.3.1.1. Direktes Model-Checking

Beim ersten Ansatz überprüfen wir, ob das Simulink-Modell selbst die von uns angegebene Spezifikation erfüllt. Dabei stützen wir uns auf die Simulationssemantik des Modells und der verwendeten Blöcke und vermeiden Codegenerierungsschritte, die die Semantik verändern können. Dadurch, dass wir nahe am Simulink-Modell arbeiten, ist es möglich Gegenbeispiele direkt im Modell anzeigen zu lassen. Ein weitere günstige Eingenschaft dieses Ansatzes ist, dass wir Modelle, die komplette Systeme mit Regler und Strecke enthalten, verifizieren können. Damit ist auch ein Schritt zu Modellierung und Verifkation hybrider Systeme (Abschnitt 2.1.3 und [Hen96]) möglich.

Den Vorteil, dass wir den Übersetzungsschritt zu C-Code nicht durchführen, können wir auch als Nachteil dieses Ansatzes sehen. Zwischen Modell und erzeugtem Code eines Reglers können Unterschiede auftreten, so dass die Verifikation der Eigenschaften des Modells keine Aussagen über die Eigenschaften des eingesetzten Codes macht. Ein weiterer Nachteil betrifft die Semantik der Modelle. Greift ein Model-Checker nicht direkt auf die Simulationssemantik zu, sondern bildet sie nach, muss er bei durch The MathWorks vorgenommenen Änderungen angepasst werden (siehe Bemerkung 2.2.3).

2.3.1.2. Model-Checking von generiertem Code

Für den zweiten Ansatz lassen wir aus dem Modell oder einem seiner Subsysteme über den Model-Checker automatisch Code erzeugen. Für diesen Code wird dann vom Model-Checker die Verifikation durchgeführt. Vorteile dieser Methode gegenüber dem direkten Model-Checking

ergeben sich, wenn der generierte Code auch produktiv eingesetzt wird. Wir erreichen damit größere Praxisnähe.

Als problematisch kann sich erweisen, dass der Model-Checker das Gegenbeispiel erst für den generierten Code erzeugt. Bringt der Model-Checker das Gegenbeispiel nicht in Bezug mit den Blöcken des Simulink-Modells, ist es für uns schwer, den Fehler im Modell zu finden. Ein weiterer Nachteil kann auftreten, wenn ein Gesamtsystem aus Regler und Strecke verifiziert werden soll, und auch aus der Strecke Code für eine Zielplattform generiert wird, was nicht erwünscht sein kann.

2.3.2. Auswahl der Model-Checker

Das wichtigste Kriterium bei der Auswahl der Model-Checker für die Evaluierung ist die Fähigkeit, Simulink-Modelle als Eingabe zu akzeptieren. Dabei erachten wir es als legitim, kleinere Anpassungen am Modell zu verlangen, wenn diese die Semantik nicht verändern. Wir betrachten im Rahmen dieser Diplomarbeit zwei kommerzielle Model-Checker, EmbeddedValidator von OSC und Safety-Checker Blockset von TNI. Beide Werkzeuge verfolgen unterschiedliche Ansätze für das Model-Checking (siehe Abschnitt 2.3.1). Laut den Angaben der Herstellen haben sie unterschiedliche Einschränkungen bezüglich der von ihnen akzeptierten Konstrukte. Im Folgenden stellen wir die beiden Werkzeuge kurz vor, wir beziehen uns dabei auf die jeweilge Dokumentation [OSC05], [TNI05].

2.3.2.1. OSC EmbeddedValidator

Bei EmbeddedValidator handelt es sich um einen symbolischen Model-Checker, der VIS[2] [BHSV+96] mit dem CUDD-BDD-Paket[3] verwendet.

EmbeddedValidator überprüft Simulink-Modelle anhand des daraus erzeugten C-Codes. Dafür benötigt er TargetLink von dSPACE (siehe Abschnitt 2.2.3. Ferner ist die in Abschnitt 1.1.3 erwähnte Software erforderlich. Bei EmbeddedValidator handelt es sich um ein eingeständiges Tool, welches die Matlab-Umgebung selbst nicht nutzt. Gestartet wird es durch den Block Simulink/Stateflow Verification Environment, den wir im Simulink-Modell plazieren.

Laut Dokumentation akzeptiert EmbeddedValidator die in Tabelle 2.3 aufgeführten Blöcke. Der Ursache dafür, dass ein Block nicht akzeptiert wird, liegt nicht immer beim EmbeddedValidator. In manchen Fällen kommt es vor, dass TargetLink einen Simulink-Block nicht übersetzen kann, und dieser dann dem EmbeddedValidator nicht zur Verfügung steht. Des weiteren beschränken EmbeddedValidator und TargetLink bei einigen Blöcken die akzeptierten Ausprägungen. In der Dokumentation sind weitere Beschränkungen aufgeführt, wie zum Beispiel dass die Verwendung von Fließkommadatentypen nicht unterstützt wird. Variablen und Signale dieser Typen müssen in passende Festkommavariablen und -signale umgewandelt werden. Um die Anforderungen zu formulieren, stehen uns alle Ein- und Ausgänge des TargetLink-Modells sowie Zustände, lokale Variablen und Ereignisse und Ein- und Ausgänge der Stateflow-Charts zur Verfügung.

Die durchführbaren Analysen umfassen neben Erreichbarkeitsanalysen auch Beweise, die vom Benutzer eingegeben werden. Als Ziele für die Erreichbarkeitsanalysen können wir Wertebereichsüberschreitungen, Zustände und Konfigurationen von Stateflow-Charts sowie benutzerdefinierte Eigenschaften eingeben.

[2] `http://embedded.eecs.berkeley.edu/Respep/Research/Vis`
[3] `http://vlsi.colorado.edu/~fabio`

ABS	Sum
Assignment	Product
Data Store Write/Memory/Read	Mux/Demux
Flip Flops	Selector
Gain	Bus Creator
Logical Operators	Bus Selector
Relational Operators	Data Type Conversion
math-functions square, reciprocal, rem, mod	For-Iterator Subsystem
Merge	While-Iterator Subsystem
Multiport Switch	Switch Case / Action Subsystem
Switch	IF-Action Subsystem
Rate Limiter	Preprocessor IF
Relay	Unit Delay Reset Enabled
Saturation	Inport/Outport
Sign	Constant
Unit-Delay	From/Goto/Goto Tag Visibility
Function-Call Generator	Ground
	Terminator

Tabelle 2.3.: Von EmbeddedValidator unterstützte Simulink-Blöcke (aus [OSC05])

Die Anforderungen der Beweise werden mittels vorgegebener Pattern definiert. Diese können weder boolesch kombiniert noch geschachtelt werden. Die Pattern definieren temporale Eigenschaften, wie zum Beispiel: „Immer wenn P erfüllt ist, ist Q in genau X Schritten erfüllt." Die Pattern sind implizit allquantifiziert. Zusätzlich können wir Annahmen über Eingangswerte und Variablen machen, die die Verifikation erleichtern.

Folgende Arbeitsschritte führen wir durch, um für ein Simulink-Modell eine Spezifikation mit EmbeddedValidator zu verifizieren.

1. Wir fügen die Blöcke TargetLink Main Dialog und Simulink/Stateflow Verification Environment auf der obersten Modellebene ein.

2. Das zu verifizierende Simulink-Subsystem lassen wir automatisch in ein TargetLink-System übersetzen.

3. Das übersetzte Subsystem wird für den EmbeddedValidator ausgewählt.

4. EmbeddedValidator erzeugt mit TargetLink automatisch C-Code.

5. Ist die Bedienschnittstelle von EmbeddedValidator gestartet, formulieren wir unsere Spezifikationen und lassen diese verifizieren.

6. Eventuelle Gegenbeispiele lassen wir uns auf Wunsch anzeigen; zur Auswahl stehen zwei Darstellungsformen:

   - Entwicklung der EmbeddedValidator zugänglichen Werte und
   - Simulation des Gegenbeispiels in Simulink.

2.3.2.2. TNI Safety-Checker Blockset

Safety-Checker Blockset arbeitet direkt auf Simulink-Modellen. Das Simulink-Modell muss nicht in eine andere Form überführt werden. Wie der Name andeutet, handelt es sich bei diesem Model-Checker nicht um eine eigenständige Software, sondern um eine Toolbox für Simulink, die eine Anzahl von spezifischen Simulink-Blöcken enthält. Safety-Checker Blockset ist für die Verifkation auf die Matlab-Umgebung angewiesen.

Die Blöcke der Toolbox benutzen wir, um Spezifikationen zu formulieren. Eine Spezifikation besteht aus Simulink-Blöcken und einem der SCB-Beweis-Blöcke. Bei der Formulierung der Spezifikation greifen wir auf Signalleitungen des Simulink-Modells zu. Zusätzlich enthält die Toolbox Blöcke, die Annahmen über Signale und Eigenschaften machen.

Die Beweis-Blöcke akzeptieren nur boolesche Signale, sie entsprechen den Pattern bei EmbeddedValidator. Mit ihnen lassen sich allquantifizierte Eigenschaften ausdrücken, wie zum Beispiel: „Nie ist Eingang 2 wahr, nachdem Eingang 1 wahr war, und bevor Eingang 3 wahr ist.“ Wie bei den Pattern des EmbeddedValidators können wir die Blöcke weder kombinieren noch schachteln.

Laut der Informationen in der Dokumentation von Safety-Checker Blockset gibt es drei Möglichkeiten, wie Blöcke behandelt werden können:

- Der Block wird semantisch korrekt in ein internes Format übertragen. Es kann sein, dass der Block bei der Verifikation trotzdem abstrahiert wird.
- Der Block und das ihn enthaltene Subsystem wird nicht akzeptiert.
- Der Block wird als Attrappe akzeptiert. Die Semantik des Blocks wird nicht beachtet, und der Block wird abstrahiert.

Bei der Abstraktion wird der interne Zustand des Blocks ignoriert, und die Ausgangssignale können den gesamten Wertebereich ihres Datentyps annehmen. In der Dokumentation befindet sich eine Auflistung darüber wie die einzelnen Blöcke behandelt werden.

Der Dokumentation entnehmen wir auch, dass manche Blöcke nur bis zu einer bestimmten Maximalzahl von Eingängen akzeptiert werden. Diese Zahl liegt oft bei 5 Eingängen. Alle Eingänge müssen vom gleichen Datentyp sein.

Signale werden abhängig von ihrem Datentyp behandelt.

- Boolesche Skalare werden akzeptiert und genau berechnet.
- Skalare Ganzzahlsignale können in ihrem Wertebereich beschränkt werden. In diesem Fall werden sie laut Dokumentation genau berechnet.
- Skalare Fließkommasignale werden wie unbeschränkte Ganzzahlsignale abstrahiert. Vektorsignale werden prinzipiell abstrahiert.

Folgende Arbeitsschritte führen wir für die Verifkation mit Safety-Checker Blockset durch.

1. Wir formulieren die Spezifikation mit Simulink-Blöcken in einem Subsystem.
2. An den Ausgang dieses Subsystems plazieren wir den passenden Beweis-Block.
3. Die Verifikation wird in der Matlab-Umgebung durchgeführt.
4. Wir lassen uns ein eventuelles Gegenbeispiel anzeigen; es werden die Belegungen der Simulink-Signale und eventuell vorhandener Stateflow-Variablen angezeigt.

2.3.2.3. Sonstige Model-Checker

Es existiert noch eine Reihe von frei verfügbaren Model-Checkern, die für die Evaluierung im Rahmen dieser Diplomarbeit nicht in Frage kommen.

CheckMate, an der Carnegie Mellon University entwickelt, ist ein Model-Checker für hybride Systeme. Er ist in Matlab implementiert und verifiziert Simulink-Modelle. Die Simulink-Umgebung dient dabei lediglich als Rahmen für die Modellierung hybrider Systeme. CheckMate bringt seine eigene Blockbibliothek mit, die sich stark an der Modellierung hybrider Systeme orientiert. Wir können in CheckMate keine gewöhnlichen Simulink-Modelle zur Verifikation eingeben. Hinzu kommt, dass CheckMate in den letzten Jahren kaum mehr weiterentwickelt und gepflegt wird.

Weitere Model-Checker wie UppAal, SMV, SPIN, etc. betrachten wir nicht weiter, da sie alle eine eigene Eingabesprache haben. Die Entwicklung einer Übersetzungsmethode würde über den Rahmen dieser Diplomarbeit hinausgehen.

3. Anforderungen an Model-Checker für Matlab/Simulink

Nachdem wir die Grundlagen von Matlab, Simulink und Stateflow sowie zwei Model-Checker kennengelernt haben, formulieren wir nun funktionale und nicht-funktionale Anforderungen für Model-Checker für Simulink.

In diesem Abschnitt verwenden wir die Begriffe „MUSS“, „DARF NICHT“, „ERFORDERLICH“, „SOLL“, „SOLL NICHT“, „EMPFOHLEN“, „KANN“ und „EMPFOHLEN“ wie in RFC 2119[1] (siehe Anhang A.4) definiert. In Tabelle 3.1 sind die hier verwendeten Übersetzungen der Begriffe zur Beschreibung der Anforderungsniveaus aufgeführt. Verwenden wir einen dieser Begriffe im Sinne von RFC 2119, steht er in Versalien geschrieben.

Englisch	Deutsch
MUST	MUSS
MUST NOT	DARF NICHT
REQUIRED	ERFORDERLICH
SHALL	MUSS
SHALL NOT	DARF NICHT
SHOULD	SOLL
SHOULD NOT	SOLL NICHT
RECOMMENDED	EMPFOHLEN
MAY	KANN
OPTIONAL	OPTIONAL

Tabelle 3.1.: Übersetzung der Beschreibungen der Anforderungsniveaus aus RFC 2119.

3.1. Funktionale Anforderungen

Bei den funktionalen Anforderungen konzentrieren wir uns auf die syntkatischen Fähigkeiten von Model-Checkern. Wichtig ist, dass sich aus dem geplanten Einsatz des Model-Checkers möglichst wenig Einschränkungen für die Modellierung ergeben. Der Model-Checker soll daher einen möglichst großen Umfang der Simulink-Syntax akzeptieren. Neben dem Modell benötigt der Model-Checker eine Spezifikation, die das Modell erfüllen soll. Zusammengefasst können wir folgende funktionale Anforderungen umreißen:

- *Der Model-Checker soll eine gewisse Auswahl von Blöcken und deren Ausprägungen akzeptieren.*
- *Er soll mit den Simulink-Datentypen umgehen können.*

[1] `http://www.ietf.org/rfc/rfc2119.txt`

- *Wir wollen Simulink-Subsysteme, Stateflow-Charts sowie S-Functions mit eigenem Code einsetzen können.*
- *Es soll möglich sein, dem Model-Checker gewisse typische Spezifikationen zu übergeben.*

In den folgenden Abschnitten konkretisieren wir diese Anforderungen.

3.1.1. Grundkonstrukte

In diesem Teil behandeln wir die Anforderungen zusammen, die Blöcke und deren Ausprägungen betreffen. Wir konzentrieren uns auf die Verwendung von Simulink beim Entwurf von eingebetteten Systemen und der dazugehörden Software. Wir erwarten aus dem Grund nicht, dass der Model-Checker mit mathematischen Konstrukten wie Ableitungen oder Integralen umgehen kann. Auch verzichten wir auf Blöcke, die kontinuierliche Zustände vorraussetzen. Wir nehmen an, dass solche Blöcke den Zustandsraum eines Modells so stark aufblähen, dass die Verifikation in akzeptabler Zeit nicht mehr möglich ist (siehe Abschnitt 2.1.4).

Wir führen den Begriff korrekt akzeptieren ein.

Definition 3.1.1. Der Model-Checker **akzeptiert** einen Block (ein Subsystem, ein Modell) **korrekt**, wenn

1. der Block syntaktisch erkannt und vom Model-Checker als Eingabe akzeptiert wird und
2. die Semantik des Blockes im Model-Checker erhalten bleibt.

Das heißt, wir können den Block dem Model-Checker eingeben, und er verhält sich bei der Verifikation im Model-Checker so wie in der Simulink-Simulation.

Bevor wir als Zwischenschritt konkretisierte Anforderungen formulieren, führen wir noch den Begriff des eigenständigen Blocks ein.

Definition 3.1.2. Einen Simulink-Block nennen wir **eigenständig**, wenn er weder als maskiertes Subsystem, noch als maskierte S-Function implementiert ist.

Nach dieser Definition sind nicht-maskierte Subsysteme und S-Functions auch eigenständige Blöcke.

Wir konkretisieren nun die die Blöcke betreffenden Anforderungen.

1. ERFORDERLICH ist, dass der Model-Checker eigenständige Blöcke korrekt akzeptiert,
   a) mit denen wir konstante Werte ausgeben können,
   b) mit denen wir aus mehreren Signalen eines auswählen können,
   c) mit denen wir Signale zusammenfassen und zusammengefasste trennen können,
   d) mit denen wir Signalleitungen abschließen können,
   e) mit denen wir Ein- und Ausgänge von Modellen und Subsystemen definieren können,
   f) mit denen wir algebraische Schleifen auflösen können.
2. ERFORDERLICH ist, dass der Model-Checker eigenständige Blöcke in der von Simulink vorgegebenen Ausprägung korrekt akzeptiert,

a) mit denen wir logische Funktionen berechnen können
b) mit denen wir einfache mathematische Operationen durchführen können,
c) mit denen wir Signalwerte vergleichen können,
d) mit denen wir aus mehreren festen Werten eines auswählen können,
e) mit denen wir Subsysteme bilden können.

3. EMPFOHLEN ist, dass der Model-Checker eigenständige Blöcke, die er in der von Simulink vorgegebenen Ausprägung korrekt akzeptiert, in einigen weiteren Ausprägungen korrekt akzeptiert.

4. EMPFOHLEN ist, dass der Model-Checker eigenständige Blöcke korrekt akzeptiert,

a) mit denen wir Signalwerte halten können,
b) mit denen wir eigene Funktionen definieren können,
c) mit denen wir variable Werte ausgeben können.

5. OPTIONAL ist, dass der Model-Checker Blöcke, die maskierte Subsysteme und maskierte S-Functions sind, korrekt akzeptiert.

6. OPTIONAL ist, dass der Model-Checker weitere eigenständige Blöcke korrekt akzeptiert.

Die sich daraus ergebenden konkreten Anforderungen tragen wir in Tabelle 3.2 ein. Folgendes Beispiel verdeutlicht, wie wir nach der Tabelle Anforderungen formulieren.

Beispiel 3.1.3. Wir formulieren die Anforderungen, die den Block Logical Operator betreffen:

1. Der Model-Checker MUSS den Block Logical Operator mit dem Operator `NOT` korrekt akzeptieren.

2. Der Model-Checker KANN den Block Logical Operator mit jedem der Operatoren `AND`, `OR`, `NAND`, `NOR` und `XOR` und einem Eingang korrekt akzeptieren.

3. Der Model-Checker MUSS den Block Logical Operator mit dem Operator `AND` und zwei Eingängen korrekt akzeptieren.

4. Der Model-Checker SOLL den Block Logical Operator mit dem Operator `AND` und mehr als zwei Eingängen korrekt akzeptieren.

...

7. Der Model-Checker SOLL den Block Logical Operator mit dem Operator `NAND` und zwei Eingängen korrekt akzeptieren.

8. Der Model-Checker KANN den Block Logical Operator mit dem Operator `NAND` und mehr als zwei Eingängen korrekt akzeptieren.

...

Die Anforderungen für den Operator `OR` gleichen den Anforderungen 3 bis 4. Weitere Anforderungen für die Operatoren `NOR` und `XOR` sind analog zu den Anforderungen 7 bis 8.

Bei einigen Blöcken können wir jedem Eingang einen Operator zuweisen (Sum, Product, ...). Bei diesen Blöcken drücken wir mit dem Eintrag *beliebig* in der Spalte Ausprägungen aus, dass zu jeder Operatorkombination eine Anforderung formuliert wird.

Kategorie	Block	Ausprägung	MUSS	SOLL	KANN
Continuous	*alle*	*alle*			✔
Discontinuities	*alle*	*alle*			✔
Discrete	Memory	*alle*	✔		
	Unit Delay	*alle*		✔	
	Zero-Order Hold	*alle*		✔	
	sonstige	*alle*			✔
Logic and Bit Operations	Logical Operator	`NOT`	✔		
		`AND, OR, NAND, NOR, XOR; 1 Input`			✔
		`AND, OR; 2 Inputs`	✔		
		`AND, OR;` *sonstige*		✔	
		`NAND, NOR, XOR; 2 Inputs`		✔	
		`NAND, NOR, XOR;` *sonstige*			✔
	Relational Operator	*alle*	✔		
Lookup Tables	Lookup Table	`Use Input Nearest`	✔		
		`Use Input` *sonstige*		✔	
		sonstige			✔
	Lookup Table (2-D)	`Use Input Nearest`	✔		
		sonstige `Use Input`		✔	
		sonstige			✔
Math Operations	Abs	*alle*	✔		
	Gain	`element-wise`	✔		
		sonstige		✔	
	Math Function	`pow, sqrt, mod`	✔		
		`log, log10`		✔	
		sonstige			✔
	MinMax	`2 inputs`	✔		
		sonstige		✔	
	Product	`element-wise; 2 Inputs,` *beliebig*	✔		
		`element-wise;` *sonstige*		✔	
		`Matrix`			✔
	Rounding Function	`round`	✔		
		`ceil, floor`		✔	
		`fix`			✔
	Sign	*alle*	✔		
		`rectangular; 2 Inputs,` *beliebig*	✔		

Fortsetzung nächste Seite

Kategorie	Block	Ausprägung	MUSS	SOLL	KANN
	Sum	`rectangular;` *sonstige*		✔	
		`round; 2 Inputs,` *beliebig*	✔		
		`round;` *sonstige*		✔	
	Trigonometric Function	`cos, sin`	✔		
		`sonstige`			✔
Model Verification	*alle*	*alle*			✔
Model-Wide Utilities	*alle*	*alle*			✔
Ports & Subsystems	Enabled Subsystem	*alle*	✔		
	Inport	*alle*	✔		
	Outport	*alle*	✔		
	Subsystem	*alle*	✔		
	Triggered Subsystem	*alle*	✔		
	sonstige	*alle*			✔
Signal Attributes	*alle*	*alle*			✔
Signal Routing	Demux	*alle*	✔		
	Multiport Switch	`zero-based indexing off`	✔		
		`zero-based indexing on`		✔	
	Mux	*alle*	✔		
	Switch	*alle*	✔		
	sonstige	*alle*			✔
Sinks	Outport	*alle*	✔		
	Terminator	*alle*	✔		
	sonstige	*alle*			✔
Sources	Constant	*alle*	✔		
	Ground	*alle*	✔		
	Inport	*alle*	✔		
	Pulse Generator	*alle*		✔	
	Sine Generator	*alle*		✔	
	Step	*alle*		✔	
	sonstige	*alle*			✔
User-Defined Functions	S-Function	*alle*		✔	
	sonstige	*alle*			✔
Additional Math & Discrete	*alle*	*alle*			✔

Tabelle 3.2.: Anforderungen, Akzeptanz von Simulink-Blöcken

Datentyp	MUSS	SOLL	KANN
uint8	✔		
uint16	✔		
uint32	✔		
int8	✔		
int16	✔		
int32	✔		
single			✔
double			✔
boolean	✔		

Tabelle 3.3.: Anforderungen, Akzeptanz von Simulink-Datentypen

3.1.2. Datentypen

Für die Codeerzeugung für eingebettete Systeme sind vor allem Ganzzahldatentypen von großer Bedeutung. Wir ordnen den sie betreffenden Anforderungen eine hohes Niveau zu. Boolean basiert in Simulink auf uint8, deswegen ist die Akzeptanz dieses Datetyps nicht so wichtig. Wir stellen vier Anforderungen bezüglich der Akzeptanz von Datentypen.

1. ERFORDERLICH ist, dass der Model-Checker alle Ganzzahldatentypen akzeptiert.
2. EMPFOHLEN ist, dass der Model-Checker boolean akzeptiert.
3. OPTIONAL ist, dass Model-Checker Fließkommadatentypen akzeptiert.
4. EMPFOHLEN ist, dass der Model-Checker Modelle mit unterschiedlichen Datentypen akzeptiert.

Die ersten drei Anforderungen ergeben die Einträge in Tabelle 3.3.

3.1.3. Diagrammtypen

Komplexere Funktionen können wir in Simulink mit drei verschiedenen Diagrammtypen implementieren, als

- Simulink-Subsystem,
- Stateflow-Chart oder
- S-Function.

Simulink-Subsysteme als Block haben wir bereits in Abschnitt 3.1.1 behandelt, dort aber keine weiteren Konstrukte eingefügt. Es ist erforderlich, dass ein Model-Checker sie richtig behandelt, da sonst eine Strukturierung des Modells unmöglich ist.

Stateflow-Charts sind für die Modellierung von ereignisgesteuerter diskreter Systeme, endlicher Automaten, etc. wichtig. Da solche Systeme den Entwurf eingebetteter Systeme erleichtern, sind Stateflow-Charts ebenfalls erforderlich.

Auch wenn die Einbindung eigenen Codes über S-Functions wünschenswert erscheint, erachten wir die Verifkation solcher Diagramme nicht als erforderliche Fähigkeit eines Model-Checkers für Simulink. Es ist zu erwarten, dass beliebiger Code nicht verifiziert werden kann.

Wir formulieren nun die Anforderungen, die Diagrammtypen betreffen.

Diagrammtyp	MUSS	SOLL	KANN
Simulink	✔		
Stateflow	✔		
S-Function		✔	

Tabelle 3.4.: Anforderungen, Akzeptanz von Simulink-Diagrammtypen

- ERFORDERLICH ist, dass der Model-Checker Simulink-Subsysteme korrekt akzeptiert.
- ERFORDERLICH ist, dass der Model-Checker Stateflow-Charts korrekt akzeptiert.
- EMPFOHLEN ist, dass der Model-Checker S-Functions korrekt akzeptiert.
- EMPFOHLEN ist, dass der Model-Checker Modelle, die Kombinationen aus Simulink-Subsystemen, Stateflow-Charts und S-Functions enthalten, korrekt akzeptiert.

Als Zusammenfassung erhalten wir Tabelle 3.4.

3.1.4. Spezifikationen

Die Anforderungen bezüglich der Spezifikationen unterteilen wir nach zwei Fragen:

1. Welche Probleme können wir dem Model-Checker eingeben?
2. Welche Eigenschaften eines Simulink-Modells können wir für die Spezifikation nutzen?

Spezifikationen, die oft an Softwaresysteme gestellt werden, sind Erreichbarkeit von Zuständen, Liveness und Safety. Aus diesem Grund stellt die Möglichkeit, Spezifikationen für Erreichbarkeitsanalysen, eine Grund- aber nicht die Hauptfunktion eines Model-Checkers dar. Die Hauptfunktion ist es, das Systemverhalten gegen eine allgemeine Spezifikation zu verifizieren. Dazu gehört, dass wir Abläufe des Systemverhaltens spezifizieren können. Dieses wird mit Temporallogiken wie CTL und LTL möglich. Eine Erweiterung davon sind Spezifikationen in TCTL, mit denen wir auf die (Simulations-)Zeit als Variable zugreifen können.

Innerhalb eines Simulink-Modells liegen verschiedene Variablen vor, die sich für die Formulierung der Spezifikation eignen. Als Mindestanforderungen verlangen wir, dass wir die Spezifkationen über Ein- und Ausgänge vom Modellen, bzw. von Subsystemen formulieren können. Allgemein wünschen wir uns den Zugriff auf Signalwerte zwischen den einzelnen Blöcken. Ob uns interne Zustände von Simulink-Blöcken zur Verfügung stehen, ist ein Zusatzmerkmal. Nützlich ist das Zusatzmerkmal, auf Zustände von Stateflow-Charts zugreifen zu können. Für Spezifikationen in TCTL muss uns zusätzlich die Simulationszeit zur Verfügung stehen.

Es ergeben sich folgende Anforderungen:

1. ERFORDERLICH ist, dass wir Erreichbarkeitsprobleme formulieren können.
2. ERFORDERLICH ist, dass wir alle Spezifikationen formulieren können, mit denen wir den Verlauf des Systemverhaltens beschreiben können.
3. OPTIONAL ist, dass wir Spezifikationen formulieren können, die sich auf die Zeit beziehen.
4. ERFORDERLICH ist, dass wir auf Werte von Signalen zugreifen können.

Problem	MUSS	SOLL	KANN
Erreichbarkeit von Systemzuständen	✔		
Verlauf des Systemverhaltens	✔		
Verhalten bezüglich Zeit			✔
Eigenschaft	**MUSS**	**SOLL**	**KANN**
Signalwerte	✔		
Ein-/Ausgänge von Subsystemen	✔		
Stateflow-Zustände		✔	
Zustände von Blöcken			✔
Simulationszeit			✔

Tabelle 3.5.: Anforderungen, Formulierung der Spezifikation

5. ERFORDERLICH ist, dass wir auf Werte von Ein- und Ausgängen von Subsystemen zugreifen können.
6. EMPFOHLEN ist, dass wir auf Zustände in Stateflow-Charts zugreifen können.
7. OPTIONAL ist, dass wir auf interne Zustände von Simulink-Blöcken zugreifen können.
8. OPTIONAL ist, dass wir auf die Simulationszeit zugreifen können.

Tabelle 3.5 fasst die Anforderungen zusammen.

3.2. Nicht-funktionale Anforderungen

Zusätzlich zu den funktionalen formulieren wir nicht-funktionale Anforderungen für Model-Checker für Simulink. Diese umfassen Qualitätsanforderungen und Anforderungen an den Entwicklungsprozess.

3.2.1. Qualitätsanforderungen

Matlab, Simulink und Stateflow stehen für folgende Plattformen zur Verfügung:

- HP-UX 11
- Linux
- Linux x86-64
- Mac OS X
- Solaris
- Windows

Wir wollen den Model-Checker unter allen Betriebssystemen nutzen, für die Matlab verfügbar ist.

Matlab und Simulink werden in verschiedenen Einsatzbereichen von Regelungstechnik bis Biologie und Finanzmodellierung verwendet. Die Bedienung des Model-Checkers muss deswegen

so gestaltet sein, dass sie auch Nichtinformatikern keine Probleme bereitet. Das bezieht sich insbesondere auf die Eingabe von Spezifikation und Modell. Sollten große Änderungen am Modell oder manuelle Übersetzungsschritte nötig sein, erschwert das den Einsatz des Model-Checkers im modellbasierten Entwurf. Auch die Ausgabe der Beweisergebnisse, vor allem des Gegenbeispiels muss für den Benutzer klar sein. Es ist oft nicht hilfreich, wenn lediglich die Variablenwerte, die in der Spezifikation vorkommen, ausgegeben werden.

Ein Problem beim Model-Checking sind große Zustandsräume, die hohen Laufzeit- und Speicherbedarf nach sich ziehen. Da der Model-Checker auch während des Entwurfsprozesses eingesetzt werden soll, wünschen wir keinen langen Laufzeiten auf kleinen Modellen oder einfachen Spezifikationen. Das langfristiges Ziel ist es, Modelle, die für den praktischen Einsatz entworfen worden sind, innerhalb einiger Stunden verifizieren zu können. An dieser Stelle werden wir keine konkreten Anforderungen zur Effizienz formulieren, da uns dafür die „Messwerkzeuge“ fehlen. Wir benötigen Modelle, die sich an der Praxis orientieren, aber in jeweils einer Komplexitätsdimension skalieren lassen. Diese Dimensionen sind:

- Anzahl und Datentyp der Variablen
- Komplexität der Spezifikation
- Komplexität der sich aus dem Modell ergebenden Funktion

Es ergeben sich folgende Qualitätsanforderungen:

1. ERFORDERLICH ist, dass der Model-Checker unter Windows und mindestens einem weiteren Betriebsystem, für das Simulink verfügbar ist, lauffähig ist.
2. EMPFOHLEN ist, dass der Model-Checker unter allen Betriebssystemen, für die Simulink verfügbar ist, lauffähig ist
3. ERFORDERLICH ist, dass der Benutzer das zu verifizierende Modell oder Subsystem nicht manuell in die Eingabesprache des Model-Checkers übersetzen muss.
4. EMPFEHLENSWERT ist, dass der Benutzer zur Vorbereitung des zu verifizierenden Modells oder Subsystems maximal 2 Blöcke einfügen muss.
5. OPTIONAL ist, dass der Benutzer zur Vorbereitung keine Änderungen am zu verifizierenden Modell oder Subsystem vornehmen muss.
6. ERFORDERLICH ist, dass der Benutzer die Spezifikation in
   - (T)CTL oder LTL oder
   - Simulink-Subsystemen oder
   - auf einer der natürlichen Sprache ähnlichen Form

   formulieren kann.
7. EMPFEHLENSWERT ist, dass der Benutzer die Spezifikation in Simulink-Subsystemen oder auf einer der natürlichen Sprache ähnlichen Form vornehmen kann.
8. OPTIONAL ist, dass der Besucher unter mehreren Eingabemöglichkeiten für die Spezifikation wählen kann.

9. ERFORDERLICH ist, dass der Model-Checker bei einem Fehlschlag des Beweises ein Gegenbeispiel anzeigen kann.

10. EMPFOHLEN ist, dass der Model-Checker zur Darstellung des Beweises nicht mehr als 10 % der Laufzeit braucht, die er für den Beweis gebraucht hat.

11. ERFORDERLICH ist, dass der Benutzer aus dem Gegenbeispiel in jedem Zeitschritt mindestens die Werte der in der Spezifikation verwendeten Variablen ablesen kann.

12. OPTIONAL ist, dass der Benutzer aus dem Gegenbeispiel in jedem Zeitschritt zusätzliche Signalwerte in dem zu verifizierenden Modell oder Subsystem ablesen kann.

13. EMPFOHLEN ist, dass bei Stateflow-Charts das Systemverhalten des Gegenbeispiels animiert dargestellt wird.

Diese Anforderungen sind nicht vollständig, da die Anforderungen zur Effizienz fehlen.

3.2.2. Anforderungen an den Entwicklungsprozess

Wir wollen den zeitlichen und finanziellen Aufwand für Installation und Einsatz des Model-Checkers gering halten. Aus diesem Grund soll der Model-Checker eigenständig und ohne zusätzliche Software außer Matlab, Simulink und Stateflow lauffähig sein.

Wir erwarten eine strukturierte Dokumentation, die neben einer Bedienungsanleitung Angaben zu Fähigkeiten und Einschränkungen des Model-Checkers und seiner Arbeitsweise enthält.

Da sich die Semantik der Simulink-Blöcke bei Versionswechseln von Simulink ändern kann, kann es sein, dass der Model-Checker nach einer Aktualisierung von Simulink nicht mehr korrekt arbeitet. Der Model-Checker benötigt unter Umständen auch eine Aktualisierung der Semantik. Der Model-Checker soll den Versionswechsel auf ein neues Matlab-Release nicht stark verzögern.

Wir fassen die Anforderungen an den Prozess zusammen:

1. EMPFOHLEN ist, dass der Model-Checker keine weitere Software benötigt, um seine volle Funktionalität zur Verfügung zu stellen.

2. ERFORDERLICH ist eine Dokumentation, die folgende Themen abdeckt:

   - Bedienungsanleitung,
   - Fähigkeiten des Model-Checkers
   - Einschränkungen des Model-Checkers der Modellierung

3. EMPFOHLEN ist eine Dokumentation, die folgende Themen abdeckt:

   - Bedienungsanleitung,
   - Fähigkeiten des Model-Checkers
   - Einschränkungen des Model-Checkers der Modellierung
   - Arbeitsweise des Model-Checkers

4. ERFORDERLICH ist eine Aktualisierung des Model-Checkers, wenn ein neues Matlab-Release vorliegt, und die Semantik der Blöcke des Model-Checkers nicht mehr mit der Simulationssemantik übereinstimmt.

5. EMPFOHLEN ist eine Aktualisierung des Model-Checkers innerhalb von einem Monat nachdem ein neues Matlab-Release vorliegt, und die Semantik der Blöcke des Model-Checkers nicht mehr mit der Simulationssemantik übereinstimmt.

4. Evaluierungssuite

In Kapitel 3 haben wir Anforderungen an Model-Checker für Simulink-Modelle gestellt. Allerdings fehlen uns Mittel, um in kurzer Zeit objektiv zu bewerten, in wie weit ein oder mehrere uns unbekannte Model-Checker diese Anforderungen erfüllen.

Die Recherche in den Dokumentationen der Model-Checker nach entsprechenden Informationen kann zum einen langwierig sein. Zum anderen ist es möglich, dass diese Informationen nicht vollständig enthalten sind. Zusätzlich fällt uns der Vergleich zwischen mehreren Model-Checkern schwer, da zu erwarten ist, dass sich der Umfang der Umfang der Dokumentationen unterscheidet. Der Aufwand, den wir bei dieser Methode treiben, steht in einem deutlichen Missverhältnis zu den zu erwartenden Ergebnissen.

Aus diesem Grund entwerfen und implementieren wir ein Werkzeug, dass uns bei der Bewertung von Model-Checkern für Simulink unterstützt.

4.1. Idee für eine Evaluierungssuite

Unser Ziel ist, einen uns unbekannten Model-Checker auf die in Kapitel 3 formulierten Anforderungen hin zu evaluieren. Wir wollen innerhalb kurzer Zeit eine objektive Bewertung erreichen. Die Vorbereitung für die Evaluierung eines Model-Checkers soll wenige Arbeitsschritte umfassen. Nachdem wir die Software installiert und uns die grundlegende Bedienung angeeignet haben, beginnnen wir mit dem Evaluierungsprozess. Eine lange Einarbeitungszeit in die Arbeitsweise des Model-Checkers ist zu vermeiden.

Ein anderes Problem ist aufgetreten, als wir die Effizienz betreffende Anforderungen formulieren wollten (Abschnitt 3.2.1). Uns fehlten Messinstrumente, um die Effizienz zu messen.

Die Idee ist, einen Satz von Simulink-Modellen, die *Evaluierungssuite*, zu erstellen. Mit dieser Evaluierungssuite wollen wir überprüfen, ob und wie ein Model-Checker die funktionalen und nicht-funktionalen Anforderungen erfüllt. Wir konzentrieren uns beim Entwurf auf zwei Bereiche der Anforderungen:

1. Funktionale Anforderungen (Abschnitt 3.1)
2. Effizienzmessung (Abschnitt 3.2.1)

Aus dieser Idee für die Evaluierungssuite ergeben sich bereits erste Anforderungen an die Suite und den Evaluierungsprozess. Zustätzliche Anforderungen betreffen die Flexibilität der Suite. Wir wollen die Suite leicht an neue Anforderungen anpassen können. Neu eingeführte syntaktische und semantische Merkmale von Simulink sollen mit geringem Aufwand in die Suite eingepflegt werden können.

Zu den Modellen in der Evaliuierungssuite gehört eine Dokumentation, die einen Leitfaden zum Evaluierungsprozess enthält. An die Evaluierungssuite stellen wir folgende Anforderungen:

1. Funktionale Anforderungen

a) ERFORDERLICH ist, dass mit der Evaluierungssuite die funktionalen Anforderungen an Model-Checker (Abschnitt 3.1) überprüfen lassen.

b) ERFORDERLICH ist, dass die Evaluierungssuite Modelle zur Bestimmung der Effizienz nach Abschnitt 3.2.1 zur Verfügung stellt.

2. Nicht-funktionale Anforderungen

a) ERFORDERLICH ist, dass wir das Ergebnis der Evaluierung des Model-Checkers unabhängig von den Herstellerangaben über die Fähigkeiten des Model-Checkers erhalten.

b) ERFORDERLICH ist, dass die Vorbereitungszeit für die Evaluierung zwei Personentage nicht überschreitet.

c) EMPFOHLEN ist, dass die Vorbereitungszeit für die Evaluierung einen Personentag nicht überschreitet.

d) ERFORDERLICH ist, dass wir das Evaluierungsergebnis für einen Model-Checker innerhalb zweier Personentage erhalten.

e) EMPFOHLEN ist, dass wir das Evaluierungsergebnis für einen Model-Checker innerhalb eines Personentages erhalten.

f) EMPFOHLEN ist, dass sich neue Modelle einfach in die Evaluierungssuite einarbeiten lassen.

g) ERFORDERLICH ist eine Dokumetation für den Evaluierungsprozess.

4.2. Implementierung

Die Evaluierungssuite ist aus drei Elementen aufgebaut:

1. den Modellen, die dem zu verifizierenden Model-Checker eingegeben werden,

2. einer eigenen Blockbibliothek, die die grundlegenden Konstrukte für die Modelle bereithält,

3. dem Vorgehensmodell, dass durch den Evaluierungsprozess leitet.

Die Modelle enthalten auf der obersten Ebene ein atomares Subsystemen und bei Bedarf Ground- und Terminator-Blöcke. Mit den Ground-, bzw. Terminator-Blöcken schließen wir Ein-, bzw. Ausgänge des Subsystems ab, um Warnungen von Simulink zu vermeiden. Diese treten bei der Simulation von Modellen mit nicht verbundenen Signalen auf. Wir verwenden einen diskreten Fixed-step-Solver einer Schrittweite von 0,01. Damit orientieren sich die Modelle an eingebetteten Systemen, bei denen auch feste Schrittweiten verwendet werden.

Das oberste Subsystem eines Modells enthält das Simulink-Diagramm, welches der Model-Checker gegen eine Spezifikation verifizieren soll. Die im Subsystem enthaltenen Konstrukte eines Suitemodells richten sich danach, welche der funktionalen Anforderungen an die Suite sie erfüllen sollen. Die Modelle teilen wir dementsprechend in zwei Module ein, Syntaxanalyse und Effizienzanalyse. Da wir viele der Subsysteme in verschiedenen Ausprägungen benötigen, erstellen wir die Modelle nicht direkt, sondern implementieren erst eine Blockbibliothek, die die parametrisierten Subsysteme enthält. Erstellen wir ein Modell für die Suite, fügen wir das benötigte Subsystem aus der Bibliothek ein und stellen dessen Parameter ein. Benötigen wir

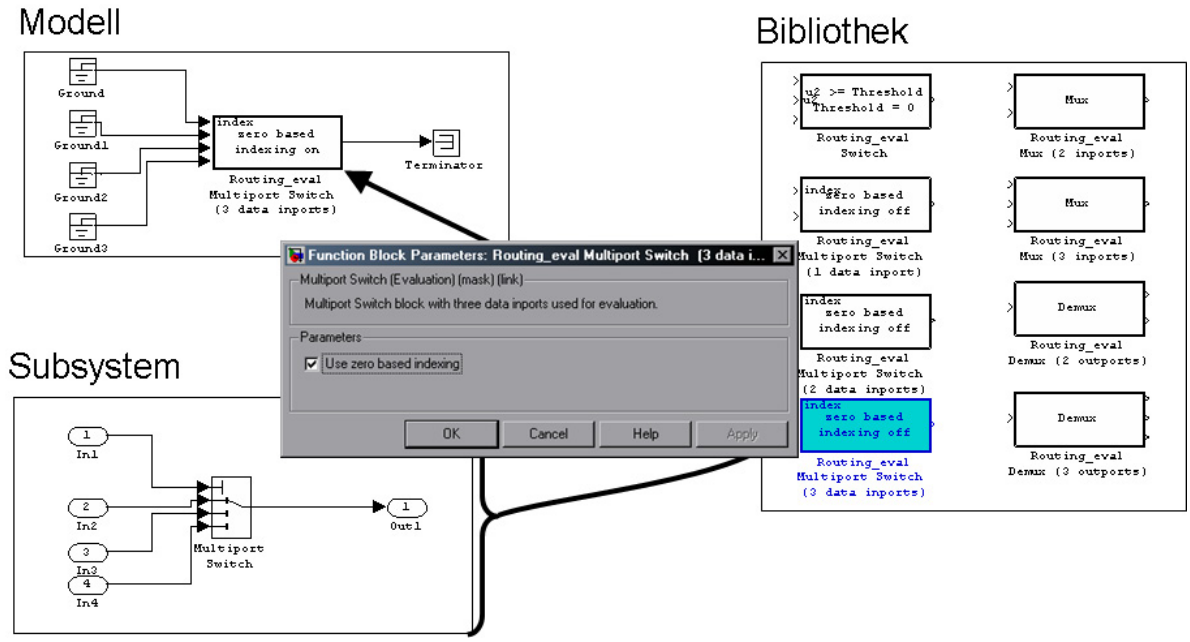


Abbildung 4.1.: Die Subsysteme mit Multiport-Switch-Block sind in der Kategorie Signal Routing der Bibliothek implementiert. Aus der Bibliothek werden sie in ein Modell eingebunden und über den Maskendialog konfiguriert.

mehrere solcher Subsysteme für ein Suitemodell, können wir diese wiederum in einem Subsystem zusammenfassen. Die Subsysteme im Modell sind mit denen in der Bibliothek verknüpft. Sind Änderungen an schon in Modellen verwendeten Subsystemen notwendig, reicht es, wenn wir sie in der Bibliothek vornehmen. Dieses Prinzip ist in Abbildung 4.1 am Beispiel eines der Subsysteme mit Multiport-Switch-Block dargestellt.

4.2.1. Blockbibliothek für die Evaluierungssuite

Die Bibliothek enthält alle in der Suite verwendeten Subsysteme. Alle Subsysteme sind atomar; so ist sichergestellt, dass ein Subsystem wie ein einzelner Block behandelt wird (siehe Abschnitt 2.2.2.1). Jedes der Subsysteme ist mit einem Maskendialog versehen, über den wir die für den Aufbau der Evaluierungssuite notwendigen Parameter des Systems einstellen. Dadurch können wir zu einem Subsystem einfach Modelle erstellen, die alle durch die Parameter möglichen Ausprägungen abdecken.

Beispiel 4.2.1. Beim Sum-Block mit fester Eingangszahl interessieren uns die Form des Blocks und die Belegung der Eingänge. Es gab ferner die Überlegung, Sum-Blöcke im Datentyp-Submodul einzusetzten. Aus diesem Grund werden nur drei Parameter über den Maskendialog 4.2angeboten, Blockform, Eingangsbelegung und Ausgangsdatentyp.

Auch wenn man bei Blöcken die Anzahl der Ein-, bzw. Ausgänge über Blockparameter einstellen kann, implementieren wir der Einfachheit halber für jede Ein-, bzw. Ausgangszahl ein Subsystem. Die korrekte Implementierung des automatischen Einsetzen, Löschen und Verbinden von In- und Outport-Blöcken erweist sich als sehr aufwändig. Die Maskendialoge dokumentieren die Funktion des Subsystems mit einem kurzen Text.

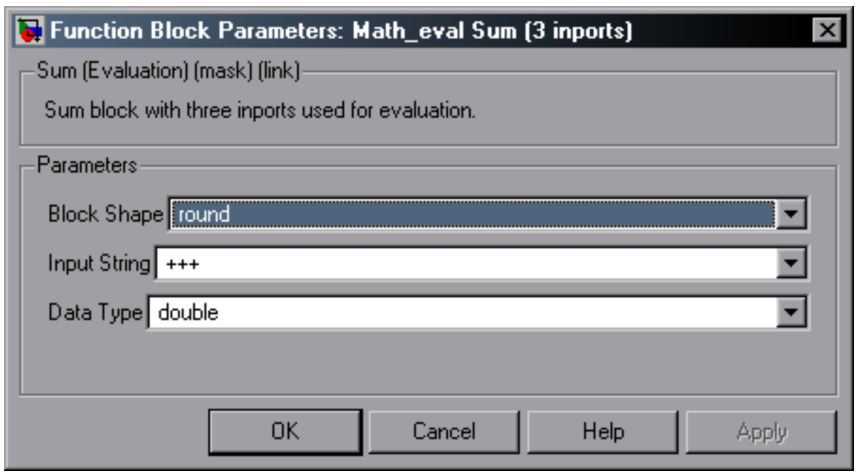


Abbildung 4.2.: Beispiel für einen der Maskendialoge der Suite-Subsysteme, Sum-Block mit drei Eingängen (vergleiche mit Abbildung 2.4 auf Seite 18)

Die Bibliothek selbst ist über Subsysteme in Kategorien unterteilt. Diese Kategorisierung spiegelt sich in der Gruppierung der Suitemodelle in Unterverzeichnissen wider. Über die Bibliothek ist es uns möglich, Änderungen in den Subsystemen rasch in die davon abhängigen Subsysteme zu propagieren.

4.2.2. Syntaxanalyse

Das erste Modul der Evaluationssuite bilden die Modelle der Syntaxanalyse. Wir untersuchen, welche Blöcke und Datentypen der Model-Checker akzeptiert. Ferner prüfen wir, ob Funktionen, die mit Kombinationen von Blöcken implementiert sind, akzeptiert werden, und ob diese auch in Stateflow oder C-Code übergeben werden können. Dafür unterteilen wir die Syntaxanalyse in drei Submodule.

4.2.2.1. Grundkonstrukte

Im ersten Submodul überprüfen wir, welche Blöcke und welche Ausprägungen der Model-Checker akzeptiert. Bei der Auswahl der Blöcke richten wir uns nach den in Abschnitt 3.1.1 formulierten Anforderungen. In unseren Modellen liegen die Blöcke in Ausprägungen vor, die die Standardversion variieren. Kann man die Zahl der Ein- oder Ausgänge eines Blocks wählen, betrachten wir neben Modellen mit dem jeweiligen Block in der Standardkonfiguration zwei, bzw. drei weitere Ausprägungen.

1. Wir überprüfen, ob Ausprägungen, die sich vom Standard unterscheiden, akzeptiert werden. Dazu erstellen wir Modelle, die den Block mit einem Eingang mehr und mit einem Eingang weniger als Standard enthalten.

2. Insbesondere überprüfen wir Blöcke mit einem Eingang. Blöcke mit mehreren möglichen Eingängen behandeln bei einem Eingangssignal dieses als Vektor und unterscheiden sich im Verhalten deswegen von der Standardvariante.

Die Modelle in diesem Submodul der Evaluierungssuite bestehen aus Subsystemen die genau einen einzelnen Block enthalten. Wir implementieren zu jedem Block möglichst alle Ausprägun-

Kategorie	Blöcke	Modellanzahl
Discontinuities	Quantizer	1
Discrete	Memory	2
	Unit Delay	2
	Zero-Order Hold	2
Logic & Bit Operations	Logical Operator	13
	Relational Operator	4
Lookup Tables	Lookup Table	5
	Lookup Table (2D)	5
Math Operations	Abs	1
	Gain	4
	Math Function	15
	MinMax	3
	Product	8
	Rounding Function	4
	Sign	1
	Sum	10
	Trigonometry	5
Ports & Subsystems	Enabled Subsystem	2
	For Iterator Subsystem	6
	Triggered Subsystem	6
	While Iterator Subsystem	3
Signal Attributes	Data Type Conversion	6
Signal Routing	Demux	2
	Multiport Switch	4
	Mux	2
	Switch	3
Sinks	Scope	1
	Stop Simulation	1
Sources	Constant	1
	Pulse Generator	2
	Signal Generator	4
	Sine Wave	2
	Step	1
Stateflow	Stateflow	4
User-Defined Functions	S-Function	1
Summe		**136**

Tabelle 4.1.: Anzahl der verwendeten Modelle in Grundkonstrukte

gen. Ist es möglich, den Ausgangsdatentyp vorzugeben, stellen wir uint8 ein. Können Zustände angegeben werden, wählen wir Null und einen beliebiegen anderen Wert.

Beispiel 4.2.2. Dem Sum-Block können wir neben der Anzahl der Eingänge ihre Belegung durch eine Rechenoperation (+ oder −) oder den Platzhalter | vorgeben. Ausserdem können wir seine Gestalt zwischen rechteckig und rund wechseln. Mit einem Eingang summiert der Sum-Block die Komponenten eines Vektors. Standard ist ein runder Block mit zwei Eingängen, die mit + belegt sind. Wir haben Modelle mit runden und rechteckigen Sum-Blöcken mit einem bis drei Eingängen erstellt, bei denen die Eingänge mit allen möglichen Operatorenkombinationen belegt sind. Ferner haben wir für jede Gestalt ein Modell mit Platzhalter und zwei Eingängen (−|+) erstellt.

Tabelle 4.1 gibt einen Überblick über die in unserem Vorgehensmodell (siehe Abschnitt 4.2.4) eingesetzten Blöcke. In Tabelle 4.1 alle möglichen Wege durch das Vorgehensmodell berücksichtigt haben, ergibt sich bei manchen Blöcken eine hohe Modellzahl. Wir haben möglichst viele Variationen implementiert, aber nicht alle in unserem Vorgehensmodell verwendet. Diese können bei Bedarf abweichend vom Vorgehensmodell verwendet werden. Im Idealfall überprüfen wir in diesem Abschnitt lediglich 60 Modelle.

4.2.2.2. Datentypen

Im zweiten Submodul überprüfen wir, welche Datentypen der Model-Checker verarbeiten kann. Dabei beschränken wir uns auf die von Simulink unterstützten Ganzzahl- und Fließkommadatentypen. Bei Bedarf können Modelle für weitere Datentypen erstellt werden.

Als Signalquellen dienen in jedem Modell ein oder mehrere Constant-Blöcke. Wir setzen sie auf vier Arten ein:

1. ein Signal (26 Modelle),
2. mehrere Signale gleichen Datentyps (9 Modelle),
3. mehrere Signale zweier Datentypen (1 Modell),
4. zwei Signale verschiedenen Datentyps (2 Modelle).

Die Anzahl der Modelle ergibt sich aus den Wertebereichsgrenzen der Datentypen. Im ersten Fall existiert für jeden Datentyp je ein Modell für jede Grenze seines Wertebereichs. Bei mehreren Signalen gleichen Datentyps werden jeweils alle Grenzen in einem Modell ausgegeben. Bei mehreren Signalen zweier Datentypen lassen wir alle Grenzen von int32 und double ausgeben. Im letzten Fall vergleichen wir mit einem Relational-Operator-Block den kleinstmöglichen double-Wert mit 0 vom Datentyp uint32. Wie schon bei den Grundkonstrukten haben wir noch weitere Modelle mit anderen Signalwerten implementiert, die bei Bedarf genutzt werden können aber in dem von uns entworfenen Vorgehensmodell nicht benötigt werden. Insgesamt liegen 81 Modelle vor.

4.2.2.3. Diagrammtypen

In Simulink können gleiche Funktionen auf verschiedene Weisen implementiert werden: mittels Simulink-Blöcken, als Stateflow-Chart oder als C-Code in einer S-Function. Das haben wir mit zwei Funktionen durchgeführt: CompareToZero und Quantizer. Für beide Funktionen liegen

jeweils vier Modelle vor, neben den drei erwähnten Arten, Funktionen zu implementieren, ein Modell mit allen drei Subsystemen. Wir überprüfen damit, ob der Model-Checker alle drei syntaktischen Möglichkeiten beherrscht.

4.2.3. Effizienzanalyse

Für die Effizienzanalyse modellieren wir Systeme, die sich leicht skalieren lassen. Gleichzeitig weisen sie keine hohe syntaktische Komplexität auf:

1. Kommuniaktion unter einfachen Automaten,
2. Inkrementieren einer Ganzzahlvariablen und
3. Auswahl eines Ausgabewerts aus Eingabewerten.

Die Spezifikationen für die einzelnen Submodule wählen wir so, dass auch sie sich in der Komplexität unterscheiden. Wir formulieren sie in natürlicher Sprache und CTL. Die einfachste Anforderung umfasst lediglich einen AG-Operator, die komplexeste schachtelt zwei Operatoren zu AGAF.

4.2.3.1. Kommunikation

Die Kommunikationsmodelle bestehen aus mehreren Automaten, die Prozesse darstellen, die sich in zwei Zuständen synchronisieren. Das geschieht, indem jeder Prozess seinen aktuellen Zustand auf ein Medium schreibt und den Zustand seines Vorgängers ausliest. Abhängig davon wird eine Transition geschaltet. Der Lese-Schreib-Zugriff auf das Medium ist mittels Monitoren realisiert. Um algebraische Schleifen zu vermeiden, verwenden wir Unit-Delay-Blöcke. Abbildung 4.3 zeigt das Modell mit 4 Automaten.

Der vom Model-Checker zu erbringende Beweis betrifft das Verhalten der Prozess-Automaten:

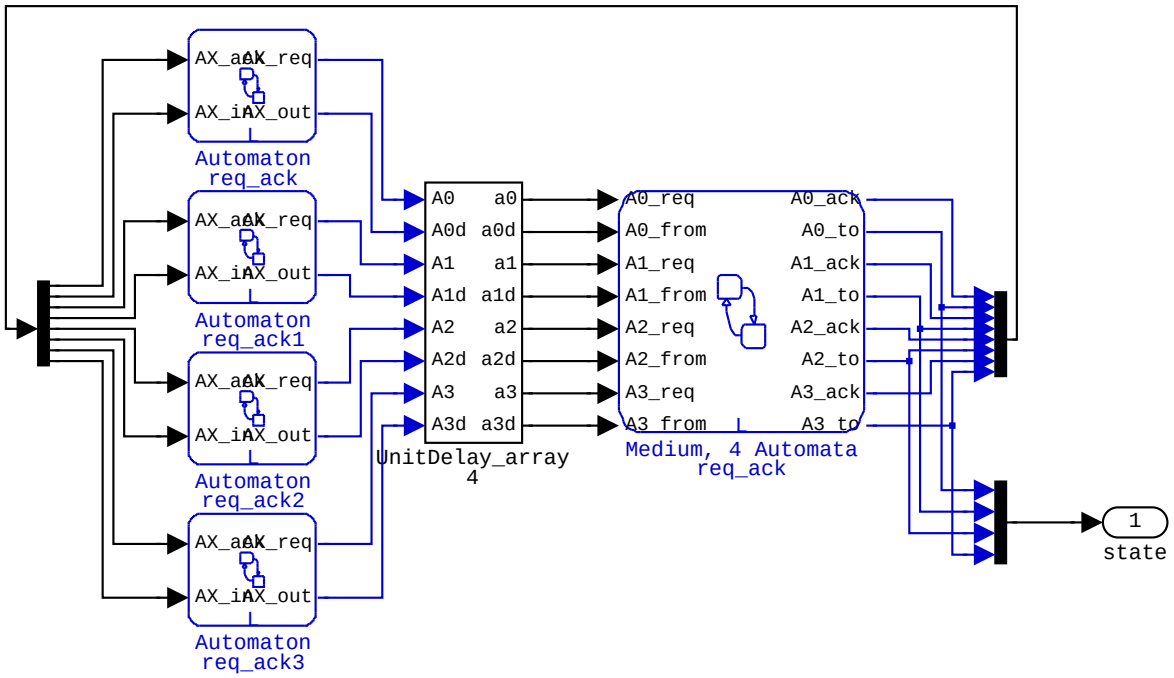
> *In jedem Zeitschritt befinden sich alle Automaten gleichzeitig in Zustand 0 und gleichzeitig in Zustand 1, und jeder Automat befindet sich immer wieder in Zustand 0 und immer wieder in Zustand 1.*

$$\begin{aligned} &\mathsf{AG}\left((A_0 = 0 \leftrightarrow \ldots \leftrightarrow A_n = 0) \wedge (A_0 = 1 \leftrightarrow \ldots \leftrightarrow A_n = 1)\right) \\ \wedge &\bigwedge_{i=0}^{n} \mathsf{AGAF}\,(A_i = 0) \wedge \bigwedge_{i=0}^{n} \mathsf{AGAF}\,(A_i = 1) \end{aligned} \tag{4.1}$$

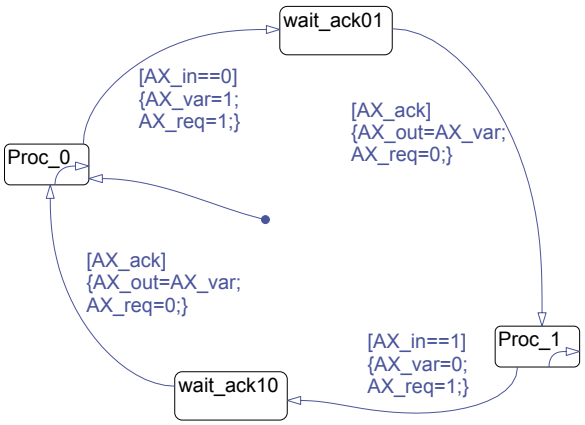
Die Zustände der Automaten für diese Spezifikation werden am Medium ausgegeben. Die Spezifikation wird von dem System erfüllt. Für jeden Prozess-Automaten vergrößert sich der Zustandsraum um den Faktor acht. Für die maximale Anzahl von zwölf Automaten ergeben sich ca. 68 Milliarden Zustände.

4.2.3.2. Zähler

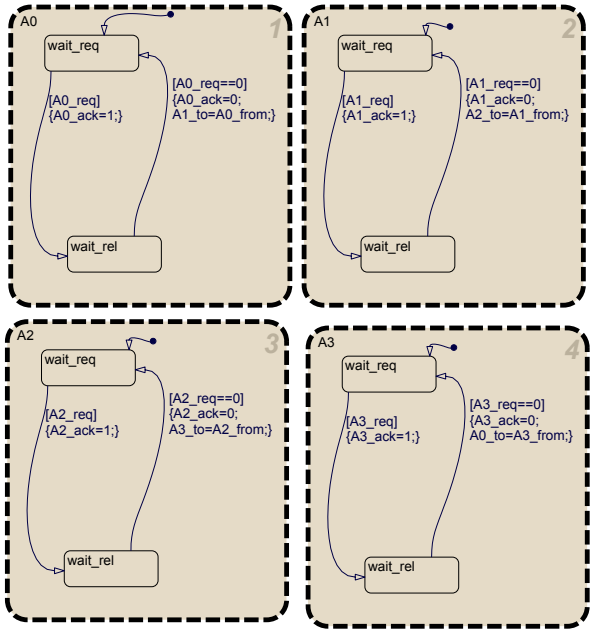
In diesem Submodul der Effizienzanalyse untersuchen wir, wie sich Model-Checker auf Zählern verhalten. Dafür wird eine Variable inkrementiert, bis eine Grenze erreicht wird. Die Wahl von Inkrement und Grenze bestimmt dabei die Modellgröße. Wir verwenden als Grenze Potenzen von Vier und wählen Eins als Inkrement. In diesem Submodul der Effizienzanalyse lässt sich die Modellgröße am einfachsten variieren.



(a) Simulink-Subsystem

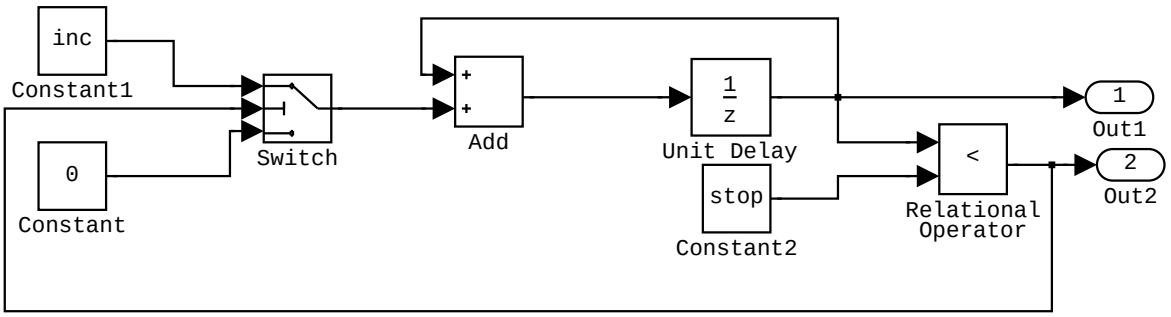


(b) Automat

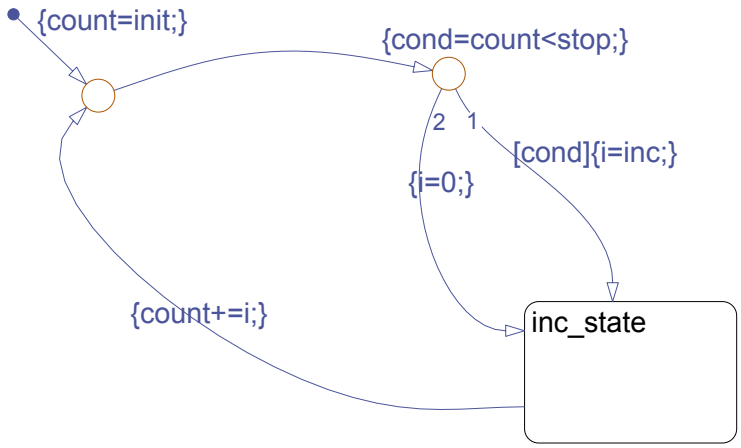


(c) Medium

Abbildung 4.3.: Beispiel für die Modelle des Kommunikationssubmoduls: vier Automaten, ein Medium



(a) Simulink



(b) Stateflow

Abbildung 4.4.: Subsysteme der Modelle des Zählersubmoduls

Wir realisieren unseren Zähler auf zwei Arten, in Simulink und in Stateflow (Abbildung 4.4). Beide Implementierungen verhalten sich auf der Simulink-Ebene gleich. So kann zusätzlich untersucht werden, bei welcher der Diagrammarten der Model-Checker effizienter arbeitet. Eine Implementierung als S-Function setzen wir nicht ein, da uns bekannt war, dass beide Model-Checker diese Blöcke nicht akzeptieren.

Wir wählen für diese Tests folgende Anforderung:

> *Es existiert ein Lauf des Systems, so dass end immer 1 ist bis counter die Grenze erreicht.*
>
> $$\mathsf{E}\,(end = 1\ \mathsf{U}\ counter = stop) \tag{4.2}$$

counter ist dabei die Zählvariable und *end* das Ergebnis des Vergleichs $counter < stop$. Die Grenze muss dabei auf jeden Fall erreicht werden. Die Spezifikation wird von allen Zählern der Effizienzanalyse erfüllt.

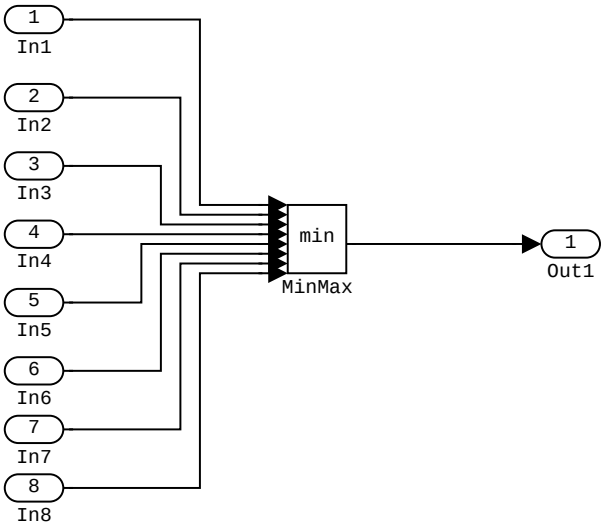


Abbildung 4.5.: Beispiel für die Modelle des Ein-/Ausgabesubmoduls: acht Eingänge

4.2.3.3. Ein- und Ausgabe

Oft werden in der Praxis ein oder mehrere Ausgangsignale aus den Eingangssignalen eines Subsystems ausgewählt. Uns interessiert, ob am Ausgang tatsächlich eines der Eingangssignale anliegt. Wie aus Vorversuchen mit dem EmbeddedValidator bekannt war, hängt die Laufzeit eines solchen Beweises stark von der Anzahl und dem Datentyp der Eingänge ab.

Die Systeme in diesem Submodul der Effizienzanalyse bestehen aus jeweils einem MinMax-Block, der aus mehreren Eingängen das Minimum ausgibt (Abbildung 4.5). Aufgrund unserer Erfahrungen mit dem EmbeddedValidator wählen wir abhängig vom Datentyp die Anzahl der Eingänge:

- 8 Modelle mit bis zu 8 uint8-Eingängen,
- 6 Modelle mit bis zu 6 uint16-Eingängen,
- 4 Modelle mit bis zu 4 uint32-Eingängen.

Es ergibt sich folgende Anforderung.

In jedem Zeitschritt ist der Ausgangswert einem der Eingangswerte gleich.

$$\mathsf{AG}\left(\bigvee_{i=1}^{n} Out = In_i\right) \tag{4.3}$$

4.2.4. Vorgehensmodell

Für den Evaluierungsprozess stellen wir ein Vorgehensmodell zur Verfügung. Dieses dient dem Nutzer der Suite als Empfehlung, welche Modelle in welcher Reihenfolge dem Model-Checker vorzulegen sind. Zweck des Vorgehensmodells ist es, die Anzahl der notwendigen Tests gering zu halten, aber dabei alle Möglichkeiten abzudecken. Aus diesem Grund überprüfen wir bei der Syntaxanalyse grundsätzlich zunächst komplexere Blockausprägungen. Bei einem Fehlschlag gehen wir zu den einfacheren über.

Beispiel 4.2.3. Der Sum-Block mit dem Eingangsbelegung $-+-$ und rechteckiger Form wird akzeptiert. Wir gehen deswegen davon aus, dass wir alle Sum-Blöcke mit drei oder zwei beliebig belegten Eingängen ($+$ oder $-$) erfolgreich eingeben können. Würde er nicht akzeptiert, ermitteln wir durch das Vorgehensmodell, welche Einstellung die Eingabe in den Model-Checker verhindert hat.

Das Vorgehensmodell ist wie die Suite modular aufgebaut. Wir können es bei Bedarf leicht um weitere Module erweitern.

Durch das Vorgehensmodell können wir den Evaluierungsprozess abkürzen. Von den Modellen im Submodul Grundkonstrukte berücksichtigen wir im Vorgehensmodell statt 210 nur noch 60 bis 137. Die Überprüfung der Datentypen können wir im optimalen Fall mit 2 Modellen durchführen, im schlechtesten Fall sind es 22 von insgesamt 38 Modellen im Vorgehensmodell. Insgesamt gibt es in diesem Submodul 81 Modelle, sie werden durch die 38 abgedeckt.

Die Ergebnisse dieser beiden Submodule der Evaluierung lassen wir im Vorgehensmodell für die weiteren Module einfließen. Wir vermeiden damit, Modelle zu überprüfen, deren syntaktischen Elemente bereits nicht akzeptiert wurden.

Das Vorgehensmodell liegt in mehreren Dokumenten vor.

1. Für den Benutzer erstellen wir einen englischsprachigen Leitfaden (`UserGuide.pdf`), der den Evaluierungsprozess kurz beschreibt.

2. Arbeitsflussdiagramme begleiten den Evaluierungsprozess:

   - Grundkonstrukte (`Basic.pdf`, Abbildungen 4.6 bis 4.8),
   - Datentypen (`DataType.pdf`, Abbildung 4.9),
   - Diagrammtypen (`DiagrammType.pdf`, Abbildung 4.10) und
   - Effizienzanalyse (`Efficiency.pdf`, Abbildung 4.11).

3. In Tabellen (Excel-Arbeitsmappe `Results.xls`) erfassen wir die Ergebnisse und lesen die Folgerungen aus erfolgreichen oder fehlgeschlagenen Tests ab.

4.2.4.1. Flussdiagramme zum Vorgehensmodell

Als Anleitung zum Evaluierungsprozess dienen die oben genannten Flussdiagramme. Deren Bestandteile sind farbkodiert.

- Grau sind Blöcke, die die Verwendung von Daten, die während der Evaluierung gewonnen wurden, repräsentieren.

- Grün sind Modelle, die auf dem Standardpfad liegen. Sie werden auf jeden Fall getestet.

- Gelb sind Modelle, die auf alternativen Pfaden liegen. Diese werden durchlaufen, wenn weiter zurückliegende Tests ergeben haben, dass der Standardpfad nicht genommen werden kann.

- Orange sind Modelle, die nach fehlgeschlagenen Tests überprüft werden.

Die Tests der Grundkonstrukte werden nach Kategorien gruppiert. Diese Testfolgen sind in dem Flussdiagramm durch Start- und End-Blöcke gekennzeichnet.

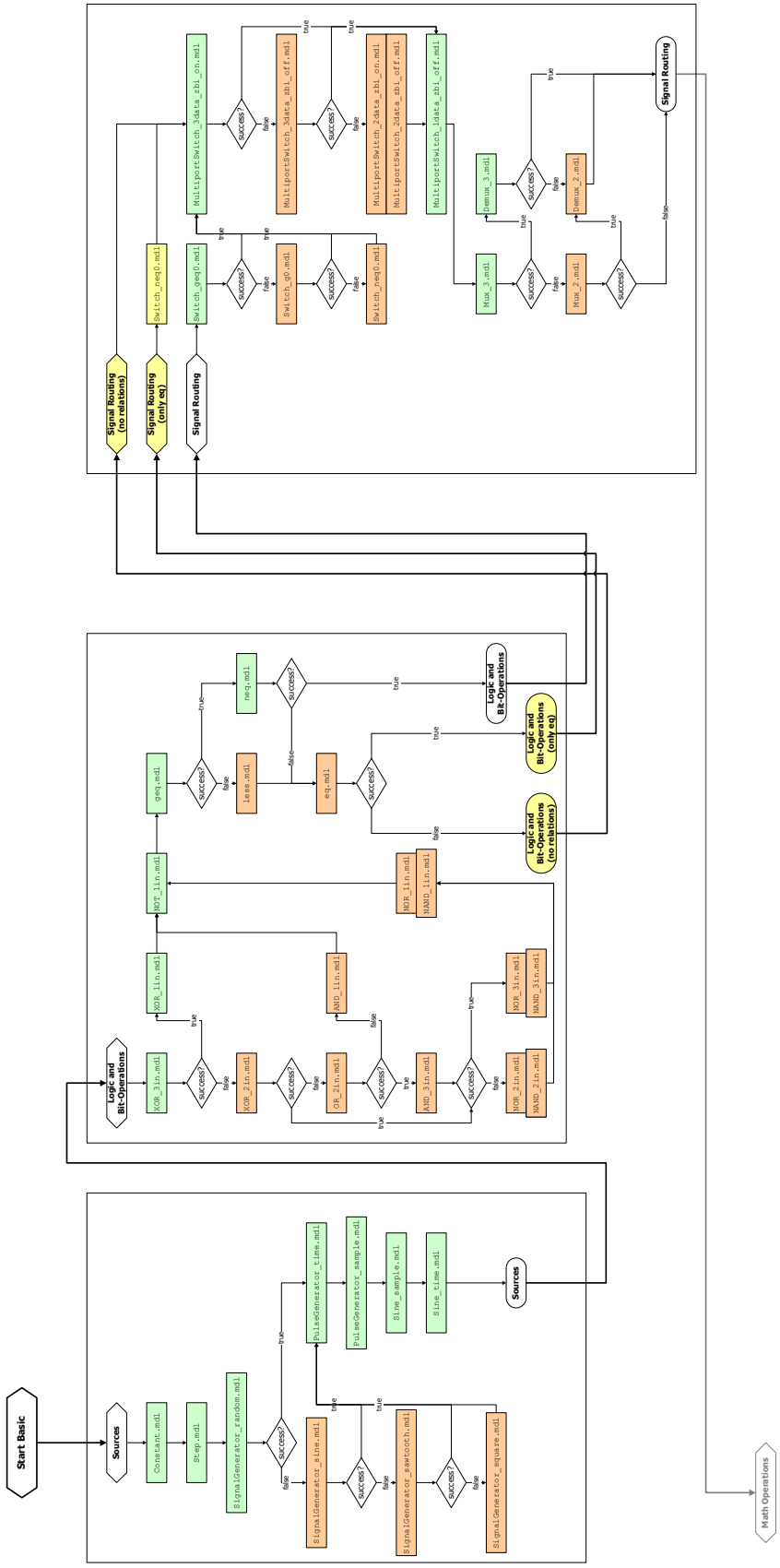


Abbildung 4.6.: Arbeitsflussdiagramm für das Modul Grundkonstrukte, Teil 1 von 3

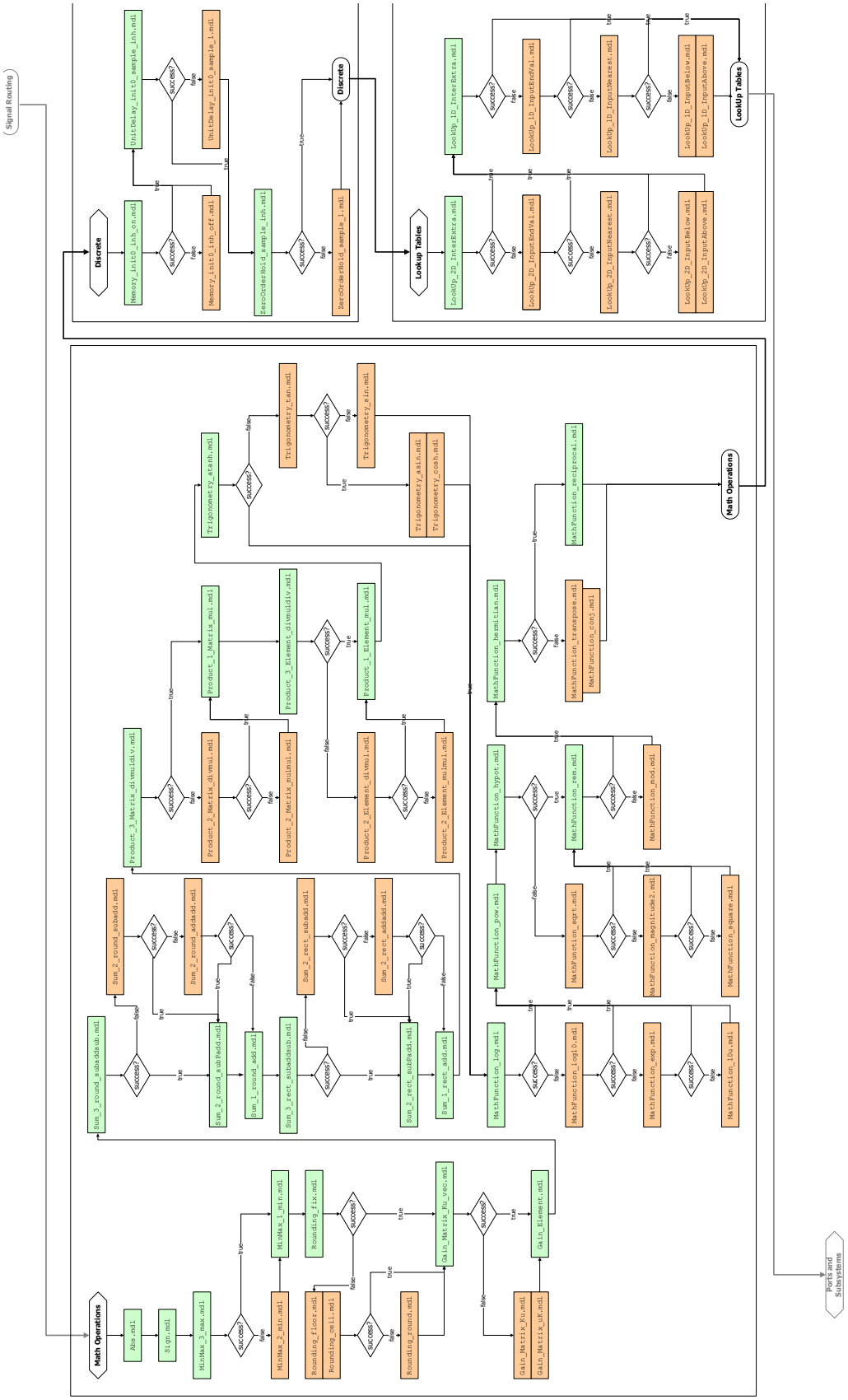


Abbildung 4.7.: Arbeitsflussdiagramm für das Submodul Grundkonstrukte, Teil 2 von 3

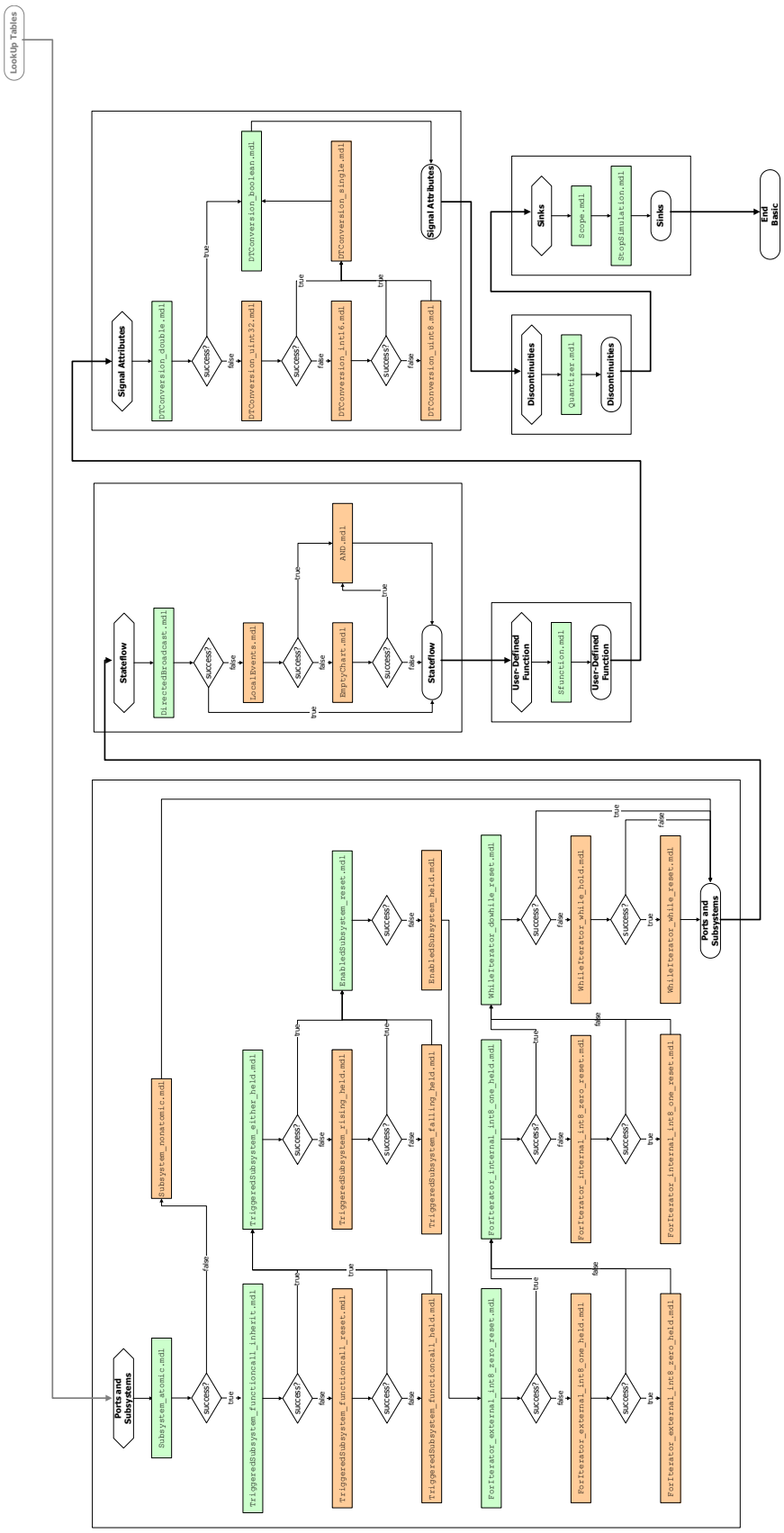


Abbildung 4.8.: Arbeitsflussdiagramm für das Submodul Grundkonstrukte, Teil 3 von 3

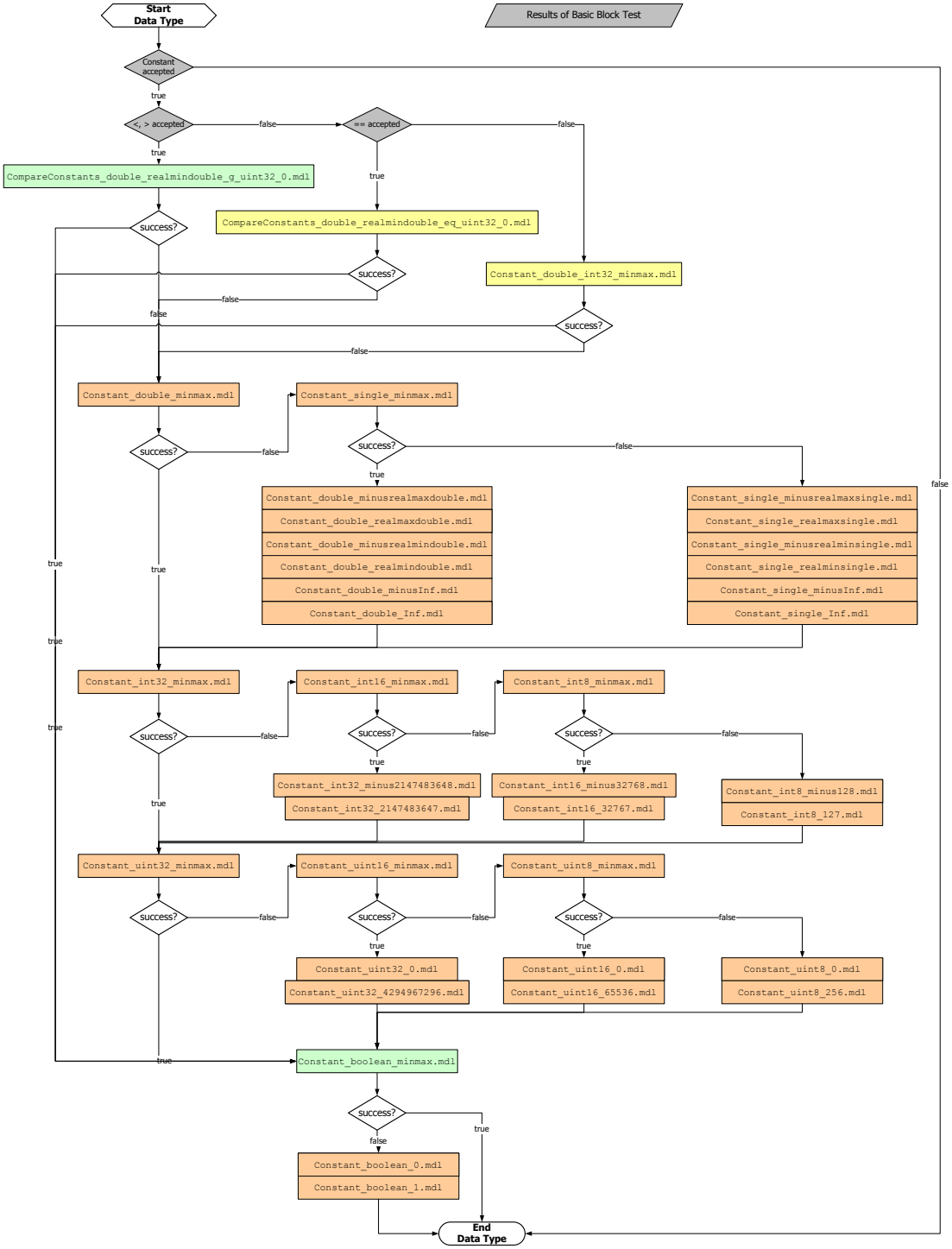


Abbildung 4.9.: Arbeitsflussdiagramm für das Submodul Datentypen

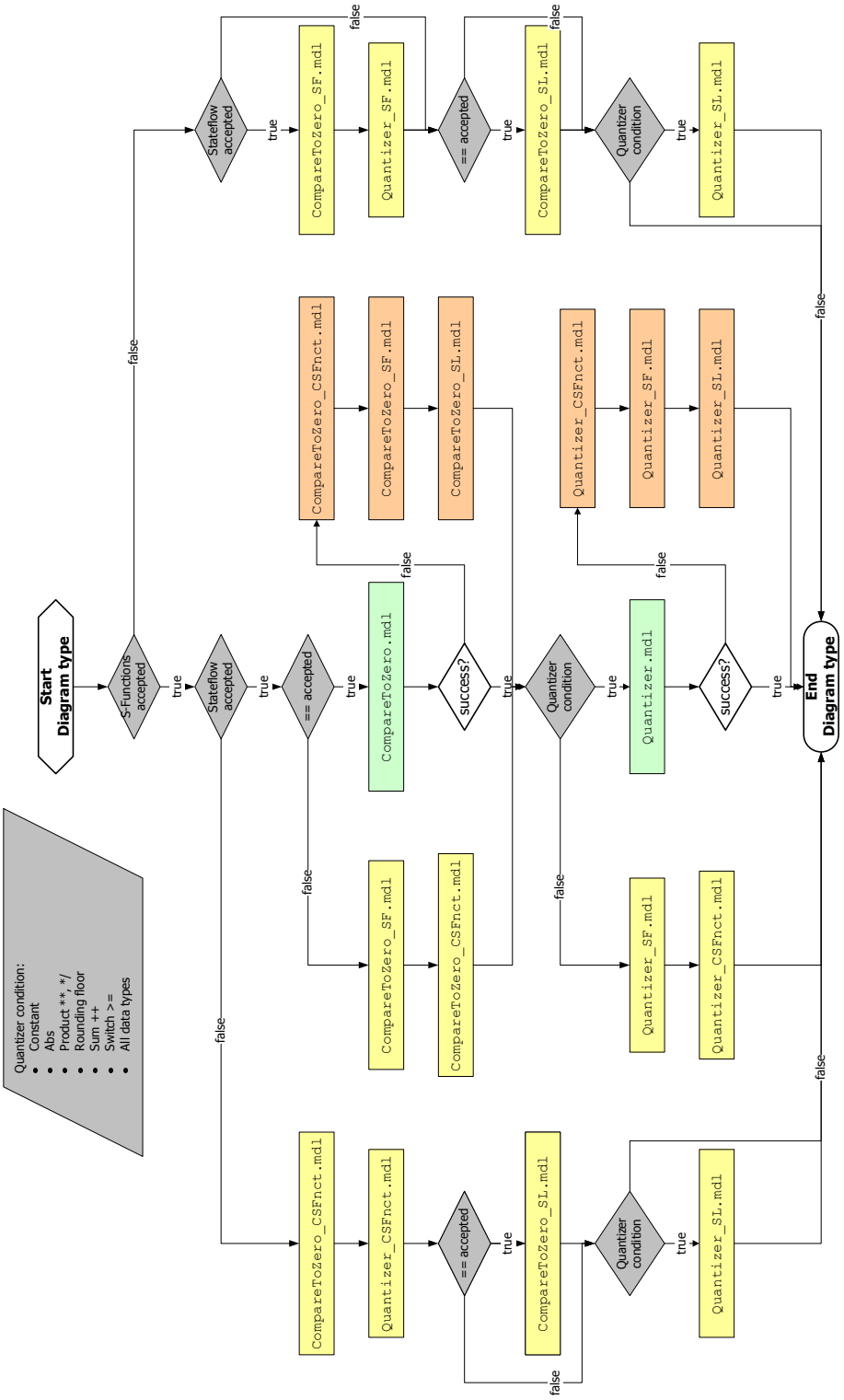


Abbildung 4.10.: Arbeitsflussdiagramm für das Submodul Diagrammtypen

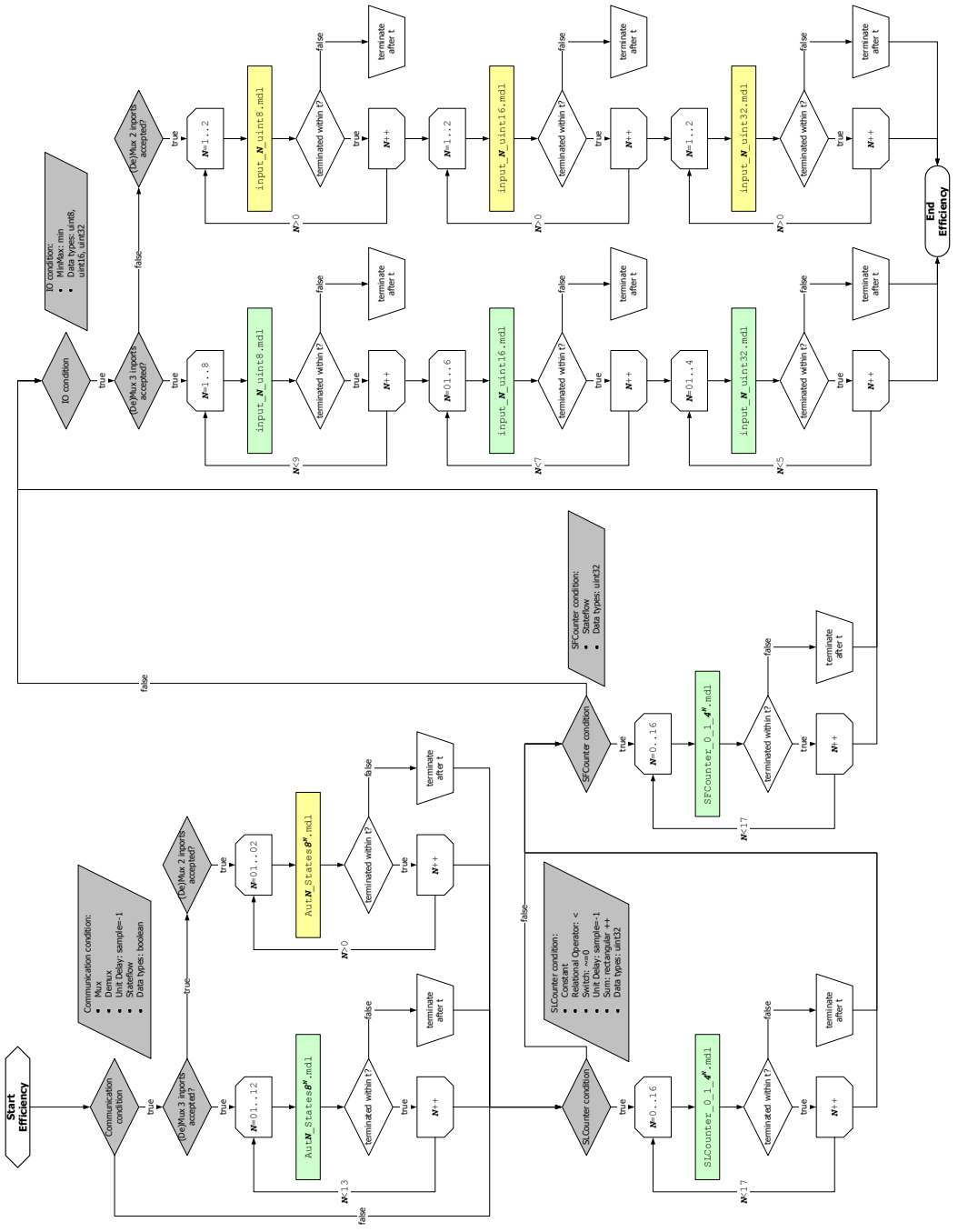


Abbildung 4.11.: Arbeitsflussdiagramm für das Submodul Effizienzanalyse

4.2.4.2. Erfassung der Ergebnisse

Die Ergebnisse der Analysen erfassen wir in Tabellen. Aus einem Erfolg oder Fehlschlag eines Test ergeben sich Schlussfolgerungen, die in diesen Tabellen ebenfalls aufgeführt werden. So können wir mehrere Modelle durch einen Test abhandeln. Die Folgerungen sind in den Tabellen 4.2 bis 4.6 aufgeführt. Bei der Syntaxanalyse ergeben sich aus erfolgreichen Tests weitere Erfolge und aus Fehlschlägen weitere Fehlschläge. Bei den Grundkonstrukten sind auch andere Kombinationen möglich (Tabellen 4.3 und 4.4).

Kategorie	Modell	Bei Erfolg: Erfolg	Bei Fehlschlag: Fehlschlag
Sources	SignalGenerator_random.mdl	SignalGenerator_sawtooth.mdl SignalGenerator_sine.mdl SignalGenerator_square.mdl	
Sources	SignalGenerator_sawtooth.mdl	SignalGenerator_square.mdl	
Sources	SignalGenerator_sine.mdl	SignalGenerator_sawtooth.mdl SignalGenerator_square.mdl	
Logic	AND_1in.mdl	OR_1in.mdl	OR_1in.mdl
Logic	AND_3in.mdl	OR_3in.mdl	OR_3in.mdl
Logic	NAND_3in.mdl	NAND_2in.mdl	NAND_2in.mdl
Logic	NOR_3in.mdl	NOR_2in.mdl	NOR_2in.mdl
Logic	OR_2in.mdl	AND_2in.mdl	AND_2in.mdl
Logic	XOR_1in.mdl	*_1in.mdl	*_1in.mdl
Logic	XOR_3in.mdl	*_2in.mdl *_3in.mdl	
Logic	geq.mdl	less.mdl greater.mdl leq.mdl	 leq.mdl
Logic	less.mdl	greater.mdl	greater.mdl
Logic	neq.mdl	eq.mdl	
Math	Gain_Matrix_Ku_vec.mdl	Gain_Matrix_Ku.mdl Gain_Matrix_uK.mdl	
Math	MathFunction_exp.mdl	MathFunction_10u.mdl	
Math	MathFunction_hermitian.mdl	MathFunction_conj.mdl MathFunction_transpose.mdl	
Math	MathFunction_hypot.mdl	MathFunction_magnitude2.mdl MathFunction_sqrt.mdl MathFunction_square.mdl	
Math	MathFunction_log.mdl	MathFunction_10u.mdl MathFunction_exp.mdl MathFunction_log10.mdl	
Math	MathFunction_log10.mdl	MathFunction_10u.mdl MathFunction_exp.mdl	
Math	MathFunction_magnitude2.mdl	MathFunction_square.mdl	
Math	MathFunction_rem.mdl	MathFunction_mod.mdl	
Math	MathFunction_sqrt.mdl	MathFunction_magnitude2.mdl MathFunction_square.mdl	
Math	MinMax_1_min.mdl	MinMax_1_max.mdl	MinMax_1_max.mdl
Math	MinMax_2_min.mdl	MinMax_2_max.mdl	MinMax_2_max.mdl
Math	MinMax_3_max.mdl	MinMax_3_min.mdl MinMax_2_max.mdl MinMax_2_min.mdl	 MinMax_3_min
Math	Product_1_Element_mul.mdl	Tabelle 4.4	Product_1_Element_div.mdl
Math	Product_1_Matrix_mul.mdl	Tabelle 4.4	Product_1_Matrix_div.mdl

Fortsetzung nächste Seite

Kategorie	Modell	Bei Erfolg: Erfolg	Bei Fehlschlag: Fehlschlag
Math	Product_2_Element_divmul.mdl	Product_2_Element_*.mdl	Product_2_Element_divdiv.mdl Product_2_Element_muldiv.mdl
Math	Product_2_Matrix_divmul.mdl	Product_2_Matrix_*.mdl	Product_2_Matrix_divdiv.mdl Product_2_Matrix_muldiv.mdl
Math	Product_3_Element_divmuldiv.mdl	Product_3_Element_*.mdl Product_2_Element_*.mdl	 Product_3_Element_*.mdl
Math	Product_3_Matrix_divmuldiv.mdl	Product_3_Matrix_*.mdl Product_2_Matrix_*.mdl	 Product_3_Matrix_*.mdl
Math	Rounding_ceil.mdl	Rounding_round.mdl	
Math	Rounding_fix.mdl	Rounding_ceil.mdl Rounding_floor.mdl Rounding_round.mdl	
Math	Rounding_floor.mdl	Rounding_round.mdl	
Math	Sum_1_rect_add.mdl	Tabelle 4.4	Sum_1_rect_sub.mdl
Math	Sum_1_round_add.mdl	Tabelle 4.4	Sum_1_round_sub.mdl
Math	Sum_2_rect_subadd.mdl	Sum_2_rect_addadd.mdl Sum_2_rect_addsub.mdl Sum_2_rect_subsub.mdl	 Sum_2_rect_addsub.mdl Sum_2_rect_subsub.mdl
Math	Sum_2_round_subadd.mdl	Sum_2_round_addadd.mdl Sum_2_round_addsub.mdl Sum_2_round_subsub.mdl	 Sum_2_round_addsub.mdl Sum_2_round_subsub.mdl
Math	Sum_3_rect_subaddsub.mdl	Sum_3_rect_*.mdl Sum_2_rect_addadd.mdl Sum_2_rect_addsub.mdl Sum_2_rect_subsub.mdl	 Sum_3_rect_*.mdl
Math	Sum_3_round_subaddsub.mdl	Sum_3_round_*.mdl Sum_2_round_addadd.mdl Sum_2_round_addsub.mdl Sum_2_round_subsub.mdl	 Sum_3_round_*.mdl
Math	Trigonometry_asin.mdl	Trigonometry_acos.mdl Trigonometry_atan.mdl	Trigonometry_acos.mdl Trigonometry_atan.mdl
Math	Trigonometry_atanh.mdl	Trigonometry_*.mdl	
Math	Trigonometry_cosh.mdl	Trigonometry_sinh.mdl Trigonometry_tanh.mdl	Trigonometry_sinh.mdl Trigonometry_tanh.mdl
Math	Trigonometry_sin.mdl	Trigonometry_cos.mdl	Trigonometry_cos.mdl
Math	Trigonometry_tan.mdl	Trigonometry_sin.mdl Trigonometry_cos.mdl	Trigonometry_a*.mdl Trigonometry_*h.mdl
Discrete	Memory_init0_inh_on.mdl	Memory_init0_inh_off.mdl	
Discrete	UnitDelay_init0_sample_1.mdl	UnitDelay_init0_sample_0.mdl	UnitDelay_init0_sample_0.mdl
Discrete	UnitDelay_init0_sample_inh.mdl	UnitDelay_init0_sample_*.mdl	
Discrete	ZeroOrderHold_sample_1.mdl	ZeroOrderHold_sample_0.mdl	ZeroOrderHold_sample_0.mdl
Discrete	ZeroOrderHold_sample_inh.mdl	ZeroOrderHold_sample_*.mdl	
Lookup	Lookup_1D_InputNearest.mdl	Lookup_1D_InputAbove.mdl	

Fortsetzung nächste Seite

Kategorie	Modell	Bei Erfolg: Erfolg	Bei Fehlschlag: Fehlschlag
		Lookup_1D_InputBelow.mdl	
Lookup	Lookup_1D_InterEndVal.mdl	Lookup_1D_InputAbove.mdl Lookup_1D_InputBelow.mdl Lookup_1D_InputNearest.mdl	
Lookup	Lookup_1D_InterExtra.mdl	Lookup_1D_InputAbove.mdl Lookup_1D_InputBelow.mdl Lookup_1D_InputNearest.mdl Lookup_1D_InterEndVal.mdl	
Lookup	Lookup_2D_InputNearest.mdl	Lookup_2D_InputAbove.mdl Lookup_2D_InputBelow.mdl	
Lookup	Lookup_2D_InterEndVal.mdl	Lookup_2D_InputAbove.mdl Lookup_2D_InputBelow.mdl Lookup_2D_InputNearest.mdl	
Lookup	Lookup_2D_InterExtra.mdl	Lookup_2D_InputAbove.mdl Lookup_2D_InputBelow.mdl Lookup_2D_InputNearest.mdl Lookup_2D_InterEndVal.mdl	
Signal Routing	Demux_3.mdl	Demux_2.mdl	
Signal Routing	MultiportSwitch_1data_zbi_off.mdl	MultiportSwitch_1data_zbi_on.mdl	
Signal Routing	MultiportSwitch_3data_zbi_off.mdl	MultiportSwitch_2data_zbi_off.mdl	
Signal Routing	MultiportSwitch_3data_zbi_on.mdl	MultiportSwitch_2data_zbi_off.mdl MultiportSwitch_2data_zbi_on.mdl MultiportSwitch_3data_zbi_off.mdl	
Signal Routing	Mux_3.mdl	Mux_2.mdl	Demux_3.mdl
Signal Routing	Switch_g0.mdl	Switch_g5.mdl Switch_neq0.mdl	 Switch_g5.mdl
Signal Routing	Switch_geq0.mdl	Switch_*.mdl	Switch_geq5.mdl
Subsystems	EnabledSubsystem_reset.mdl	EnabledSubsystem_held.mdl	
Subsystems	ForIterator_external_int8_one_held.mdl		ForIterator_external_int8_*_*.mdl
Subsystems	ForIterator_external_int8_zero_reset.mdl	ForIterator_external_int8_*_*.mdl	
Subsystems	ForIterator_internal_int8_one_held.mdl	ForIterator_internal_int8_*_*.mdl	
Subsystems	ForIterator_internal_int8_zero_reset.mdl		ForIterator_internal_int8_*_*.mdl
Subsystems	Subsystem_atomic.mdl		* Subsystems
Subsystems	Subsystem_nonatomic.mdl		* Subsystems
Subsystems	TriggeredSubsystem_either_held.mdl	TriggeredSubsystem_falling_held.mdl TriggeredSubsystem_rising_held.mdl	
Subsystems	TriggeredSubsystem_functioncall_inherit.mdl	TriggeredSubsystem_functioncall_*.mdl	
Subsystems	TriggeredSubsystem_functioncall_reset.mdl	TriggeredSubsystem_functioncall_held.mdl	
Subsystems	TriggeredSubsystem_rising_held.mdl	TriggeredSubsystem_falling_held.mdl	
Subsystems	WhileIterator_dowhile_reset.mdl	WhileIterator_*_*.mdl	
Subsystems	WhileIterator_while_held.mdl		WhileIterator_*_*.mdl
Subsystems	WhileIterator_while_reset.mdl		WhileIterator_*_reset.mdl
Stateflow	AND.mdl	EmptyChart.mdl	
Stateflow	DirectedBroadcast.mdl	AND.mdl	

Fortsetzung nächste Seite

Kategorie	Modell	Bei Erfolg: Erfolg	Bei Fehlschlag: Fehlschlag
		EmptyChart.mdl LocalEvents.mdl	
Stateflow	EmptyChart.mdl		AND.mdl
Signal Attributes	DTConversion_double.mdl	DTConversion_*int*.mdl DTConversion_single.mdl	
Signal Attributes	DTConversion_int16.mdl	DTConversion_*int16.mdl DTConversion_*int8.mdl	DTConversion_uint32.mdl
Signal Attributes	DTConversion_uint32.mdl	DTConversion_*int*.mdl	DTConversion_int32.mdl
Signal Attributes	DTConversion_uint8.mdl	DTConversion_int8.mdl	DTConversion_int8.mdl

Tabelle 4.2.: Folgerungen bei erfolgreichen, bzw. fehlgeschlagenen Tests im Submodul Grundkonstrukte, Teil 1 von 2

Kategorie	Modell	Bei Erfolg: Fehlschlag	Bei Fehlschlag: Erfolg
Signal Routing	MultiportSwitch_3data_zbi_off.mdl	MultiportSwitch_2data_zbi_on.mdl	
Subsystems	ForIterator_external_int8_zero_held.mdl	ForIterator_external_int8_one_reset.mdl	ForIterator_external_int8_one_reset.mdl
Subsystems	ForIterator_internal_int8_one_reset.mdl	ForIterator_internal_int8_zero_reset.mdl	ForIterator_internal_int8_zero_reset.mdl
Subsystems	WhileIterator_while_reset.mdl	WhileIterator_dowhile_*.mdl	

Tabelle 4.3.: Folgerungen bei erfolgreichen, bzw. fehlgeschlagenen Tests im Submodul Grundkonstrukte, Teil 2 von 2

Kategorie	Bedingung (∧-verknüpft)		Folgerung (∧-verknüpft)	
	Erfolg	Fehlschlag	Erfolg	Fehlschlag
Math	Sum_1_round_add.mdl Sum_2_round_*sub*.mdl		Sum_1_round_sub.mdl	
Math	Sum_1_round_add.mdl	Sum_2_round_*sub*.mdl		Sum_1_round_sub.mdl
Math	Sum_1_rect_add.mdl Sum_2_rect_*sub*.mdl		Sum_1_rect_sub.mdl	
Math	Sum_1_rect_add.mdl	Sum_2_rect_*sub*.mdl		Sum_1_rect_sub.mdl
Math	Product_1_Element_mul.mdl Product_2_Element_*div*.mdl		Product_1_Element_div.mdl	
Math	Product_1_Element_mul.mdl	Product_2_Element_*div*.mdl		Product_1_Element_div.mdl
Math	Product_1_Matrix_mul.mdl Product_2_Matrix_*div*.mdl		Product_1_Matrix_div.mdl	
Math	Product_1_Matrix_mul.mdl	Product_2_Matrix_*div*.mdl		Product_1_Matrix_div.mdl

Tabelle 4.4.: Ergänzungen zu Tabelle 4.2

4.2.4.3. Leitfaden zum Vorgehensmodell

Der englischsprachige Leitfaden beschreibt das Vorgehensmodell. Er behandelt die Punkte:

1. Kurzeinführung in die Suite,
2. Beschreibung der Arbeitsflussgraphen,
3. Erklärung des Erfassens der Ergebnisse,
4. Angabe der Spezifikationen für die Effizienzanalyse und
5. Vorbereitung der Suitemodelle für die Evaluierung.

Die ersten vier Punkte haben wir bereits weiter oben behandelt. Die Suitemodelle müssen vor der Eingabe in den Model-Checker vorbereitet werden.

1. Die Suite wird in ein eigenes Arbeitsverzeichnis kopiert. Dabei muss unter Umständen darauf geachtet werden, dass die Pfadnamen zu den einzelnen Modellen eventuelle Bedingungen des Model-Checkers erfüllen.
2. Das Arbeitsverzeichnis mit der Suite und das Verzeichnis, das die Bibliothek `lib_eval.mdl` enthält, müssen in Matlabs Pfad-Variable eingetragen werden.
3. In jedem Modell in der Arbeitsversion der Suite müssen vor dem Test die Verknüpfungen zu der Bibliothek aufgebrochen werden.

Je nach Model-Checker können noch weitere vorbereitende Schritte notwendig sein.

- Einfügen weiterer obligatorischer Blöcke.
- Entfernen der Masken von Subsystemen.

Spezifikationen für die Syntaxanalyse haben wir nicht fest vorgegeben. Wird ein Modell in diesem Abschnitt der Evaluierung akzeptiert, soll der Benutzer mit Hilfe einer für das jeweilige Modell typischen Spezifikation überprüfen, ob das Modell auch korrekt akzeptiert wird.

Der genaue Wortlaut des Leitfadens befindet sich in Anhang A.5.

4.3. Dokumentation

Die Dokumentation der Evaluierungssuite umfasst:

- die vorliegende Diplomarbeit,
- die Graphiken und den Leitfaden des Vorgehensmodells,
- die Beschreibung der Subsysteme in den Maskendialogen.

Modell	Bei Erfolg: Erfolg
Constant_double_int32_minmax.mdl	CompareConstants_*single*.mdl CompareConstants_*double*.mdl Constant_double_*.mdl Constant_single_*.mdl Constant_*int*_*.mdl
Constant_int8_minmax.mdl	Constant_int8_*.mdl
Constant_uint8_0.mdl	Constant_uint8_*.mdl
Constant_int16_minmax.mdl	Constant_int16_*.mdl Constant_int8_*.mdl
Constant_uint16_minmax.mdl	Constant_uint16_*.mdl Constant_uint8_*.mdl
Constant_int32_minmax.mdl	Constant_int*_*.mdl
Constant_uint32_minmax.mdl	Constant_uint*_*.mdl
Constant_boolean_minmax.mdl	Constant_boolean_*.mdl
Constant_single_minmax.mdl	Constant_single_*.mdl
Constant_double_minmax.mdl	Constant_double_*.mdl Constant_single_*.mdl
CompareConstants_double_realmindouble_eq_uint32_0.mdl	CompareConstants_*single*.mdl CompareConstants_*double*.mdl Constant_double_*.mdl Constant_single_*.mdl Constant_*int*_*.mdl
CompareConstants_double_realmindouble_g_uint32_0.mdl	CompareConstants_*single*.mdl CompareConstants_*double*.mdl Constant_double_*.mdl Constant_single_*.mdl Constant_*int*_*.mdl

Tabelle 4.5.: Folgerungen bei erfolgreichen Tests im Submodul Datentypen

Modell	Bei Erfolg: Erfolg
CompareToZero.mdl	CompareToZero_*.mdl
Quantizer.mdl	Quantizer_*.mdl

Tabelle 4.6.: Folgerungen bei erfolgreichen Tests im Submodul Diagrammtypen

5. Erprobung der Evaluierungssuite

Die Erprobung der Evaluierungssuite wird durchgeführt, indem wir sie auf zwei Model-Checker für Simulink, OSC EmbeddedValidator und TNI Safety-Checker Blockset, anwenden.

5.1. Vorbereitung der Suitemodelle

Wie schon im Leitfaden des Vorgehensmodells (Abschnitt 4.2.4.3) erwähnt, müssen wir vorbereitende Schritte vornehmen, bevor wir mit der Erprobung beginnen. Für die Model-Checker, die wir erproben, sehen diese konkret so aus:

1. Vor dem Öffnen des ersten Modells:
   a) Erstellen je einer Kopie der Evaluierungssuite pro Model-Checker; für EmbeddedValidator haben wir zusätzlich die Verzeichnisnamen verkürzt, da er eine maximale Pfadnamenlänge von 256 Zeichen hat.
   b) Aktualisieren der Matlab-Pfadvariablen; der Pfad zur Bibliothek `lib_eval.mdl` muss enthalten sein.
2. Bei jedem Modell:
   a) Lösen aller Verknüpfungen zu Bibliotheken.
   b) Hinzufügen zusätzlicher Blöcke:
      EmbeddedValidator: • TargetLink Main Dialog,
      • Simulink/Stateflow Verification Environment.
      Safety-Checker Blockset: • Anforderung,
      • Beweis-Block,
      • Ersetzten von Ground, bzw. Terminator durch In-, bzw. Outport.

5.2. Durchführung und Ergebnisse der Erprobung

Mit EmbeddedValidator wurden beide Module der Evaluierungssuite erprobt. Da Safety-Checker Blockset erst ab Ende Oktober zur Verfügung stand, wurde mit dieser Software nur die Effizienzanalyse durchgeführt.

In den Tabellen zur Syntaxanalyse sind erfolgreiche Test mit ✔ und Fehlschläge mit ✘ markiert. Bei der Effizienzanalyse erfassen wir die Laufzeiten in Sekunden. Die Beweise werden nach maximal einer Stunde abgebrochen. Im Falle eines Abbruchs werden dem Vorgehensmodell folgend keine größeren Beweise durchgeführt.

5.2.1. OSC EmbeddedValidator

Bei der Überprüfung mit EmbeddedValidator orientieren wir uns an unserem Verfahrensmodell.

5.2.1.1. Syntaxanalyse

Zusätzlich zu der Frage, ob ein Block akzeptiert wird, testen wir bei jedem akzeptierten Block mit einer Anforderung, ob der Block richtig behandelt wird. Die Anforderung ist dabei vom Block abhängig, sie beschreibt eine seiner charakteristischen Eigenschaften.

Beispiel 5.2.1. Das Modell `MultiportSwitch_3data_zbi_on.mdl` enthält einen Multiport-Switch mit 3 Dateneingängen, deren Indizierung mit 0 statt mit 1 (Simulink-Standard) beginnt. Das Subsystem wird ohne weitere Fehlermeldung von TargetLink übersetzt und vom EmbeddedValidator akzeptiert. Wir überprüfen die Korrektheit, in dem wir den Kontrolleingang, mit dem wir das auszugebende Eingangssignal auswählen, den Wert 0 fest vorgeben.
Nun spezifizieren wir $\mathsf{AG}\,(Out = In1)$ mit dem Pattern `init_P_invariant` mit `P` := `Out` == `In1`. Der Beweis schlägt fehl, da bereits der Multiport-Switch-Block von TargetLink die Indizierung der Dateneingänge mit 1 beginnen lässt.

Wichtige Ergebnisse im Submodul Grundkonstrukte sind:

1. Von TargetLink nicht akzeptierte Blöcke:
   - alle Source-Blöcke ausser Constant,
   - Product-Blöcke mit elementweiser Multiplikation mit mehr als 3 Eingängen,
   - For-Iterator-Block mit externem Inkrement,
   - Memory,
   - Quantizer.
2. Einschränkungen bei der Übersetzung in TargetLink:
   - Bei Matrix-Invertierung mit einem Product-Block müssen Fließkommadatentypen verwendet werden; diese werden von EmbeddedValidator nicht akzeptiert.
   - TargetLink akzeptiert keine Platzhalter (|) in Sum-Blöcken.
   - TargetLink kann bei der Übersetzung von runden Sum-Blöcken Signalverbindungen verlieren.
   - Die Option „zero-based indexing“ beim Multiport-Switch-Block wird von TargetLink nicht unterstützt; es erfolgt keine Warnung bei der Übersetzung.
3. Von EmbeddedValidator nicht akzeptierte Blöcke:
   - Rounding Function,
   - Trigonemtry,
   - Math Function mit allen Operatoren außer `mod`, `rem`, `reciprocal` und `square`
   - sämtliche Lookup-Table-Blöcke,
   - S-Function.
4. EmbeddedValidator kann Enable-/Trigger-Signale an Subsysteme nicht beachten.

Alle Ergebnisse dieses Submoduls der Syntaxanalyse befinden sich in Tabelle 5.1.

Aus Ergebnissen des Submoduls Grundkonstrukte ist uns bekannt, dass der EmbeddedValidator keine Fließkommadatentypen akzeptiert. Wir führen den Test des Modells `CompareConstants_double_realmindouble_g_uint32_0.mdl` trotzdem durch, um eine Bestätigung dafür zu haben. Wie erwartet schlägt er fehl. Die anschließenden Tests mit Fließkommadatentypen werden übersprungen und ebenfalls als Fehlschläge markiert. Bei erfolgreichen Tests von Modellen mit Ausgabe konstanter Ganzzahlsignale kontrollieren wir anhand der Interface-Auflistung von EmbeddedValidator, ob die Datentypen korrekt erkannt wurden. Die kompletten Ergebnisse zu den Datentyp-Modellen befinden sich in Tabelle 5.2.

Aufgrund der Ergebnisse aus dem Grundkonstrukte- und dem Datentyp-Submodul wissen wir, dass die Quantizer-Modelle des Diagrammtyp-Submoduls vom EmbeddedValidator nicht akzeptiert werden. Von den CompareToZero-Modellen wurde lediglich das Modell mit der Stateflow-Implementierung angenommen. Das Simulink-Modell besteht aus dem bei Simulink mitgelieferten Block. Dieses maskierte Subsystem wird bei der Übersetzung in TargetLink nicht akzeptiert.

Kategorie	Modell	Ergebnis	Bemerkungen
Sources	`Constant.mdl`	✔	
Sources	`PulseGenerator_sample.mdl`	✘	nicht unterstützt von TL
Sources	`PulseGenerator_time.mdl`	✘	nicht unterstützt von TL
Sources	`SignalGenerator_random.mdl`	✘	nicht unterstützt von TL
Sources	`SignalGenerator_sawtooth.mdl`	✘	nicht unterstützt von TL
Sources	`SignalGenerator_sine.mdl`	✘	nicht unterstützt von TL
Sources	`SignalGenerator_square.mdl`	✘	nicht unterstützt von TL
Sources	`Sine_sample.mdl`	✘	nicht unterstützt von TL
Sources	`Sine_time.mdl`	✘	nicht unterstützt von TL
Sources	`Step.mdl`	✘	nicht unterstützt von TL
Sinks	`Scope.mdl`	✔	
Sinks	`StopSimulation.mdl`	✘	nicht unterstützt von TL
Logic	`AND_1in.mdl`	✔	
Logic	`AND_2in.mdl`	✔	
Logic	`AND_3in.mdl`	✔	
Logic	`NAND_1in.mdl`	✔	
Logic	`NAND_2in.mdl`	✔	
Logic	`NAND_3in.mdl`	✔	
Logic	`NOR_1in.mdl`	✔	
Logic	`NOR_2in.mdl`	✔	
Logic	`NOR_3in.mdl`	✔	
Logic	`NOT_1in.mdl`	✔	
Logic	`OR_1in.mdl`	✔	
Logic	`OR_2in.mdl`	✔	
Logic	`OR_3in.mdl`	✔	
Logic	`XOR_1in.mdl`	✔	Eingangsignal: Vektor der Länge 2
Logic	`XOR_2in.mdl`	✔	
Logic	`XOR_3in.mdl`	✔	
Logic	`eq.mdl`	✔	
Logic	`geq.mdl`	✔	
Logic	`greater.mdl`	✔	
Logic	`leq.mdl`	✔	
Logic	`less.mdl`	✔	

Fortsetzung nächste Seite

Kategorie	Modell	Ergebnis	Bemerkungen
Logic	neq.mdl	✔	
Math	Abs.mdl	✔	
Math	Gain_Element.mdl	✔	
Math	Gain_Matrix_Ku.mdl	✔	
Math	Gain_Matrix_Ku_vec.mdl	✔	
Math	Gain_Matrix_uK.mdl	✔	
Math	MathFunction_10u.mdl	✘	nicht unterstützt von EV
Math	MathFunction_conj.mdl	✘	nicht unterstützt von EV
Math	MathFunction_exp.mdl	✘	nicht unterstützt von EV
Math	MathFunction_hermitian.mdl	✘	nicht unterstützt von EV
Math	MathFunction_hypot.mdl	✘	nicht unterstützt von EV
Math	MathFunction_log.mdl	✘	nicht unterstützt von EV
Math	MathFunction_log10.mdl	✘	nicht unterstützt von EV
Math	MathFunction_magnitude2.mdl	✘	nicht unterstützt von EV
Math	MathFunction_mod.mdl	✔	
Math	MathFunction_pow.mdl	✘	nicht unterstützt von EV
Math	MathFunction_reciprocal.mdl	✔	
Math	MathFunction_rem.mdl	✔	
Math	MathFunction_sqrt.mdl	✘	nicht unterstützt von EV
Math	MathFunction_square.mdl	✔	
Math	MathFunction_transpose.mdl	✘	nicht unterstützt von EV
Math	MinMax_1_max.mdl	✔	
Math	MinMax_1_min.mdl	✔	
Math	MinMax_2_max.mdl	✔	
Math	MinMax_2_min.mdl	✔	
Math	MinMax_3_max.mdl	✔	
Math	MinMax_3_min.mdl	✔	
Math	Product_1_Element_div.mdl	✔	
Math	Product_1_Element_mul.mdl	✔	
Math	Product_1_Matrix_div.mdl	✔	
Math	Product_1_Matrix_mul.mdl	✘	
Math	Product_2_Element_divdiv.mdl	✔	
Math	Product_2_Element_divmul.mdl	✔	

Fortsetzung nächste Seite

Kategorie	Modell	Ergebnis	Bemerkungen
Math	`Product_2_Element_muldiv.mdl`	✔	
Math	`Product_2_Element_mulmul.mdl`	✔	
Math	`Product_2_Matrix_divdiv.mdl`	✘	
Math	`Product_2_Matrix_divmul.mdl`	✘	siehe `Product_3_Matrix_divmuldiv.mdl`
Math	`Product_2_Matrix_muldiv.mdl`	✘	
Math	`Product_2_Matrix_mulmul.mdl`	✔	
Math	`Product_3_Element_divdivdiv.mdl`	✘	
Math	`Product_3_Element_divdivmul.mdl`	✘	
Math	`Product_3_Element_divmuldiv.mdl`	✘	TL/EV akzeptiert max. 2 Eingänge bei Codegenerierung
Math	`Product_3_Element_divmulmul.mdl`	✘	
Math	`Product_3_Element_muldivdiv.mdl`	✘	
Math	`Product_3_Element_muldivmul.mdl`	✘	
Math	`Product_3_Element_mulmuldiv.mdl`	✘	
Math	`Product_3_Element_mulmulmul.mdl`	✘	
Math	`Product_3_Matrix_divdivdiv.mdl`	✘	
Math	`Product_3_Matrix_divdivmul.mdl`	✘	
Math	`Product_3_Matrix_divmuldiv.mdl`	✘	TL verlangt Fließkommadaten bei Matrix und div EV akzeptiert keine Fließkommadatentypen
Math	`Product_3_Matrix_divmulmul.mdl`	✘	
Math	`Product_3_Matrix_muldivdiv.mdl`	✘	
Math	`Product_3_Matrix_muldivmul.mdl`	✘	
Math	`Product_3_Matrix_mulmuldiv.mdl`	✘	
Math	`Product_3_Matrix_mulmulmul.mdl`	✔	siehe `Product_2_Matrix_mulmul.mdl`
Math	`Rounding_ceil.mdl`	✘	nicht unterstützt von EV
Math	`Rounding_fix.mdl`	✘	nicht unterstützt von EV
Math	`Rounding_floor.mdl`	✘	nicht unterstützt von EV
Math	`Rounding_round.mdl`	✘	nicht unterstützt von EV
Math	`Sign.mdl`	✔	
Math	`Sum_1_rect_add.mdl`	✔	
Math	`Sum_1_rect_sub.mdl`	✔	
Math	`Sum_1_round_add.mdl`	✔	
Math	`Sum_1_round_sub.mdl`	✔	
Math	`Sum_2_rect_addadd.mdl`	✔	

Fortsetzung nächste Seite

Kategorie	Modell	Ergebnis	Bemerkungen
Math	Sum_2_rect_addsub.mdl	✔	
Math	Sum_2_rect_subPadd.mdl	✘	|nicht akzeptiert von TL
Math	Sum_2_rect_subadd.mdl	✔	
Math	Sum_2_rect_subsub.mdl	✔	
Math	Sum_2_round_addadd.mdl	✔	
Math	Sum_2_round_addsub.mdl	✔	
Math	Sum_2_round_subPadd.mdl	✘	|nicht akzeptiert von TL
Math	Sum_2_round_subadd.mdl	✔	
Math	Sum_2_round_subsub.mdl	✔	
Math	Sum_3_rect_addaddadd.mdl	✔	
Math	Sum_3_rect_addaddsub.mdl	✔	
Math	Sum_3_rect_addsubadd.mdl	✔	
Math	Sum_3_rect_addsubsub.mdl	✔	
Math	Sum_3_rect_subaddadd.mdl	✔	
Math	Sum_3_rect_subaddsub.mdl	✔	
Math	Sum_3_rect_subsubadd.mdl	✔	
Math	Sum_3_rect_subsubsub.mdl	✔	
Math	Sum_3_round_addaddadd.mdl	✔	TL kann Signale aufbrechen
Math	Sum_3_round_addaddsub.mdl	✔	TL kann Signale aufbrechen
Math	Sum_3_round_addsubadd.mdl	✔	TL kann Signale aufbrechen
Math	Sum_3_round_addsubsub.mdl	✔	TL kann Signale aufbrechen
Math	Sum_3_round_subaddadd.mdl	✔	TL kann Signale aufbrechen
Math	Sum_3_round_subaddsub.mdl	✔	TL kann Signale aufbrechen
Math	Sum_3_round_subsubadd.mdl	✔	TL kann Signale aufbrechen
Math	Sum_3_round_subsubsub.mdl	✔	TL kann Signale aufbrechen
Math	Trigonometry_acos.mdl	✘	nicht unterstützt von EV
Math	Trigonometry_acosh.mdl	✘	nicht unterstützt von EV
Math	Trigonometry_asin.mdl	✘	nicht unterstützt von EV
Math	Trigonometry_asinh.mdl	✘	nicht unterstützt von EV
Math	Trigonometry_atan.mdl	✘	nicht unterstützt von EV
Math	Trigonometry_atanh.mdl	✘	nicht unterstützt von EV
Math	Trigonometry_cos.mdl	✘	nicht unterstützt von EV
Math	Trigonometry_cosh.mdl	✘	nicht unterstützt von EV

Fortsetzung nächste Seite

Kategorie	Modell	Ergebnis	Bemerkungen
Math	`Trigonometry_sin.mdl`	✘	nicht unterstützt von EV
Math	`Trigonometry_sinh.mdl`	✘	nicht unterstützt von EV
Math	`Trigonometry_tan.mdl`	✘	nicht unterstützt von EV
Math	`Trigonometry_tanh.mdl`	✘	nicht unterstützt von EV
Discrete	`Memory_init0_inh_off.mdl`	✘	nicht unterstützt von TL
Discrete	`Memory_init0_inh_on.mdl`	✘	nicht unterstützt von TL
Discrete	`UnitDelay_init0_sample_0.mdl`	✔	
Discrete	`UnitDelay_init0_sample_1.mdl`	✔	
Discrete	`UnitDelay_init0_sample_inh.mdl`	✔	
Discrete	`ZeroOrderHold_sample_0.mdl`	✔	
Discrete	`ZeroOrderHold_sample_1.mdl`	✔	
Discrete	`ZeroOrderHold_sample_inh.mdl`	✔	
Lookup	`Lookup_1D_InputAbove.mdl`	✘	nicht unterstützt von EV
Lookup	`Lookup_1D_InputBelow.mdl`	✘	nicht unterstützt von EV
Lookup	`Lookup_1D_InputNearest.mdl`	✘	nicht unterstützt von EV
Lookup	`Lookup_1D_InterEndVal.mdl`	✘	nicht unterstützt von EV
Lookup	`Lookup_1D_InterExtra.mdl`	✘	nicht unterstützt von EV
Lookup	`Lookup_2D_InputAbove.mdl`	✘	nicht unterstützt von EV
Lookup	`Lookup_2D_InputBelow.mdl`	✘	nicht unterstützt von EV
Lookup	`Lookup_2D_InputNearest.mdl`	✘	nicht unterstützt von EV
Lookup	`Lookup_2D_InterEndVal.mdl`	✘	nicht unterstützt von EV
Lookup	`Lookup_2D_InterExtra.mdl`	✘	nicht unterstützt von EV
Signal Routing	`Demux_2.mdl`	✔	
Signal Routing	`Demux_3.mdl`	✔	
Signal Routing	`MultiportSwitch_1data_zbi_off.mdl`	✔	
Signal Routing	`MultiportSwitch_1data_zbi_on.mdl`	✘	TL dunterstützt zero based indexing nicht keine Fehlermeldung
Signal Routing	`MultiportSwitch_2data_zbi_off.mdl`	✔	
Signal Routing	`MultiportSwitch_2data_zbi_on.mdl`	✘	TL unterstützt zero based indexing nicht keine Fehlermeldung
Signal Routing	`MultiportSwitch_3data_zbi_off.mdl`	✔	
Signal Routing	`MultiportSwitch_3data_zbi_on.mdl`	✘	TL unterstützt zero based indexing nicht keine Fehlermeldung
Signal Routing	`Mux_2.mdl`	✔	

Fortsetzung nächste Seite

Kategorie	Modell	Ergebnis	Bemerkungen
Signal Routing	`Mux_3.mdl`	✔	
Signal Routing	`Switch_g0.mdl`	✔	
Signal Routing	`Switch_g5.mdl`	✔	
Signal Routing	`Switch_geq0.mdl`	✔	
Signal Routing	`Switch_geq5.mdl`	✔	
Signal Routing	`Switch_neq0.mdl`	✔	
Subsystems	`EnabledSubsystem_held.mdl`	(✔)	EV beachtet Trigger/Enable-Signal nicht
Subsystems	`EnabledSubsystem_reset.mdl`	(✔)	EV beachtet Trigger/Enable-Signal nicht
Subsystems	`ForIterator_external_int8_one_held.mdl`	✘	nicht akzeptiert von TL
Subsystems	`ForIterator_external_int8_one_reset.mdl`	✘	nicht akzeptiert von TL
Subsystems	`ForIterator_external_int8_zero_held.mdl`	✘	nicht akzeptiert von TL
Subsystems	`ForIterator_external_int8_zero_reset.mdl`	✘	nicht akzeptiert von TL
Subsystems	`ForIterator_internal_int8_one_held.mdl`	✔	
Subsystems	`ForIterator_internal_int8_one_reset.mdl`	✔	
Subsystems	`ForIterator_internal_int8_zero_held.mdl`	✔	
Subsystems	`ForIterator_internal_int8_zero_reset.mdl`	✔	
Subsystems	`Subsystem_atomic.mdl`	✔	
Subsystems	`Subsystem_nonatomic.mdl`	✔	
Subsystems	`TriggeredSubsystem_either_held.mdl`	(✔)	EV beachtet Trigger/Enable-Signal nicht
Subsystems	`TriggeredSubsystem_falling_held.mdl`	(✔)	EV beachtet Trigger/Enable-Signal nicht
Subsystems	`TriggeredSubsystem_functioncall_held.mdl`	(✔)	EV beachtet Trigger/Enable-Signal nicht
Subsystems	`TriggeredSubsystem_functioncall_inherit.mdl`	(✔)	EV beachtet Trigger/Enable-Signal nicht
Subsystems	`TriggeredSubsystem_functioncall_reset.mdl`	(✔)	EV beachtet Trigger/Enable-Signal nicht
Subsystems	`TriggeredSubsystem_rising_held.mdl`	(✔)	EV beachtet Trigger/Enable-Signal nicht
Subsystems	`WhileIterator_dowhile_held.mdl`	✔	
Subsystems	`WhileIterator_dowhile_reset.mdl`	✔	
Subsystems	`WhileIterator_while_held.mdl`	✔	
Subsystems	`WhileIterator_while_reset.mdl`	✔	
Discontinuities	`Quantizer.mdl`	✘	nicht unterstützt von TL
Stateflow	`AND.mdl`	✔	
Stateflow	`DirectedBroadcast.mdl`	✔	
Stateflow	`EmptyChart.mdl`	✔	
Stateflow	`LocalEvents.mdl`	✔	

Fortsetzung nächste Seite

Kategorie	Modell	Ergebnis	Bemerkungen
User-defined	SFunction.mdl	✘	nicht unterstützt von EV
Signal Attributes	DTConversion_boolean.mdl	✔	
Signal Attributes	DTConversion_double.mdl	✔	
Signal Attributes	DTConversion_int16.mdl	✔	
Signal Attributes	DTConversion_int32.mdl	✔	
Signal Attributes	DTConversion_int8.mdl	✔	
Signal Attributes	DTConversion_single.mdl	✔	
Signal Attributes	DTConversion_uint16.mdl	✔	
Signal Attributes	DTConversion_uint32.mdl	✔	
Signal Attributes	DTConversion_uint8.mdl	✔	

Tabelle 5.1.: Ergebnisse der Syntaxanalyse (Grundkonstrukte) mit EmbeddedValidator

Modell	Erfolg	Bemerkungen
Constant_double_int32_minmax.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_int8_127.mdl	✔	
Constant_int8_minmax.mdl	✔	
Constant_int8_minus127.mdl	✔	
Constant_int8_minus128.mdl	✔	
Constant_int8_pi.mdl	✔	
Constant_uint8_0.mdl	✔	
Constant_uint8_1.mdl	✔	
Constant_uint8_255.mdl	✔	
Constant_uint8_256.mdl	✔	
Constant_uint8_minmax.mdl	✔	
Constant_uint8_pi.mdl	✔	
Constant_int16_32767.mdl	✔	
Constant_int16_minmax.mdl	✔	
Constant_int16_minus32767.mdl	✔	
Constant_int16_minus32768.mdl	✔	
Constant_int16_pi.mdl	✔	
Constant_uint16_0.mdl	✔	
Constant_uint16_1.mdl	✔	
Fortsetzung nächste Seite		

Modell	Erfolg	Bemerkungen
Constant_uint16_65535.mdl	✔	
Constant_uint16_65536.mdl	✔	
Constant_uint16_minmax.mdl	✔	
Constant_uint16_pi.mdl	✔	
Constant_int32_2147483647.mdl	✔	
Constant_int32_minmax.mdl	✔	
Constant_int32_minus2147483647.mdl	✔	
Constant_int32_minus2147483648.mdl	✔	
Constant_int32_pi.mdl	✔	
Constant_uint32_0.mdl	✔	
Constant_uint32_1.mdl	✔	
Constant_uint32_4294967295.mdl	✔	
Constant_uint32_4294967296.mdl	✔	
Constant_uint32_minmax.mdl	✔	
Constant_uint32_pi.mdl	✔	
Constant_boolean_0.mdl	✔	
Constant_boolean_1.mdl	✔	
Constant_boolean_minmax.mdl	✔	
Constant_single_0.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_single_Inf.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_single_minmax.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_single_minusInf.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_single_minusrealmaxsingle.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_single_minusrealminsingle.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_single_pi.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_single_realmaxsingle.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_single_realminsingle.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_double_0.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_double_Inf.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_double_minmax.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_double_minusInf.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_double_minusrealmaxdouble.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_double_minusrealmindouble.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_double_pi.mdl	✘	EV akzeptiert keine Fließkommadatentypen
		Fortsetzung nächste Seite

Modell	Erfolg	Bemerkungen
Constant_double_realmaxdouble.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_double_realmindouble.mdl	✘	EV akzeptiert keine Fließkommadatentypen
Constant_sfix8_test.mdl	✘	EV akzeptiert keine Fließkommadatentypen
CompareConstants_double_realmindouble_eq_single_0.mdl	✘	EV akzeptiert keine Fließkommadatentypen
CompareConstants_double_realmindouble_eq_uint32_0.mdl	✘	EV akzeptiert keine Fließkommadatentypen
CompareConstants_double_realmindouble_g_uint32_0.mdl	✘	EV akzeptiert keine Fließkommadatentypen
CompareConstants_single_pi_eq_double_pi.mdl	✘	EV akzeptiert keine Fließkommadatentypen
CompareConstants_single_pi_g_double_pi.mdl	✘	EV akzeptiert keine Fließkommadatentypen
CompareConstants_single_realmindouble_eq_single_0.mdl	✘	EV akzeptiert keine Fließkommadatentypen
CompareConstants_uint8_intmaxuint8_eq_int32_256.mdl	✔	
CompareConstants_uint8_intmaxuint8_l_int32_256.mdl	✔	

Tabelle 5.2.: Ergebnisse der Syntaxanalyse (Datentypen) mit EmbeddedValidator

Modell	Erfolg	Bemerkungen
CompareToZero_CSFnct.mdl		Test nicht durchführbar
CompareToZero_SF.mdl	✔	
CompareToZero_SL.mdl	✘	CompareToZero-Block nicht unterstützt von TL
Quantizer_CSFnct.mdl		Test nicht durchführbar
Quantizer_SF.mdl	✔	
Quantizer_SL.mdl	✘	Test nicht durchführbar
CompareToZero.mdl		Test nicht durchführbar
Quantizer.mdl		Test nicht durchführbar

Tabelle 5.3.: Ergebnisse der Syntaxanalyse (Diagrammtypen) mit EmbeddedValidator

Modell	Laufzeit [s]	Bemerkungen
Aut01_States8.mdl	11	AGAF nicht ausdrückbar
Aut02_States64.mdl	11	AGAF nicht ausdrückbar
Aut03_States512.mdl	11	AGAF nicht ausdrückbar
Aut04_States4096.mdl	11	AGAF nicht ausdrückbar
Aut05_States32768.mdl	14	AGAF nicht ausdrückbar
Aut06_States262144.mdl	16	AGAF nicht ausdrückbar
Aut07_States2097152.mdl	17	AGAF nicht ausdrückbar
Aut08_States16777216.mdl	22	AGAF nicht ausdrückbar
Aut09_States134217728.mdl	20	AGAF nicht ausdrückbar
Aut10_States1073741824.mdl	29	AGAF nicht ausdrückbar
Aut11_States8589934592.mdl	39	AGAF nicht ausdrückbar
Aut12_States68719476736.mdl	59	AGAF nicht ausdrückbar

Tabelle 5.4.: Ergebnisse der Effizienzanalyse (Kommunikation) mit EmbeddedValidator

5.2.1.2. Effizienzanalyse

Die Anforderung für die Kommunikationsmodelle lässt sich nicht vollständig in Pattern ausdrücken, da EmbeddedValidator für die CTL-Operatorkombination AGAF kein Pattern anbietet. Wir überprüfen aus diesem Grund nur den ersten Teil der Anforderung. Dafür benutzen wir das Pattern `init_P_invariant`, welches dem CTL-Operator AG entspricht. In `P` drücken wir aus, dass die Prozessor-Automaten gleichzeitig in Zustand 0 und gleichzeitig in Zustand 1 sind. Wir fragen dafür nicht die Ausgänge ab, sondern nutzen die Möglichkeit des EmbeddedValidator direkt auf Objekte in Stateflow zugreifen zu können. Die Konjunktion aus dem ersten Teil von Formel (4.1) müssen wir auflösen, was in langen Formeln für `P` mündet. Wir verzichten hier auf eine Darstellung dieser Formeln. EmbeddedValidator benötigt für die Beweise maximal 59 Sekunden (12 Automaten). Um den Beweis für die Modelle mit einem bis vier Automaten durchzuführen, benötigt EmbeddedValidator eine Grundzeit von 11 Sekunden (siehe Tabelle 5.4).

Die Anforderung für die Zählermodelle muss negiert formuliert werden. Dafür zeigen wir, dass `init_Q_onlyafter_P_immidiate` mit $\texttt{P} := \texttt{counter} == \texttt{end}, \texttt{Q} := \texttt{stop} \neq \texttt{1}$ nicht erfüllt ist. Dieses Pattern entspricht dem Release-Operator (AR), dem zu EU dualen Operator. Wir führen für die Simulink-Modelle zwei Messungen durch. In der ersten Messung erfassen wir die Zeit für den Beweis, in der zweiten Messung die akkumulierte Zeit für den Beweis und die Erzeugung und Anzeige des Gegenbeispiels. Wir brechen die Versuche jeweils nach einer maximalen Laufzeit von einer Stunde ab. Aus Zeitgründen haben wir für die Stateflow-Modelle auf die Testreihe mit Erstellung der Gegenbeispiele verzichtet. Die Tabellen 5.5 zeigen die Zeiten, die EmbeddedValidator für die Beweise gebraucht hat. Um bessere Aussagen über das Verhalten von EmbeddedValidator bei wachsender Zustandsraumgröße machen zu können, überprüfen wir zusätzlich zu den vom Vorgehensmodell vorgeschlagenen Modellen je ein zusätzliches mit dem Endwert 32758. Abbildung 5.1 zeigt das Verhalten der Laufzeit in Abhängigkeit von der Zustandsraumgröße. Drei Eigenschaften des Verhaltens fallen uns auf.

1. Ab der Zählergrenze 256 wächst die Laufzeit exponentiell.
2. Die Stateflow-Modelle werden geringfügig langsamer abgearbeitet.

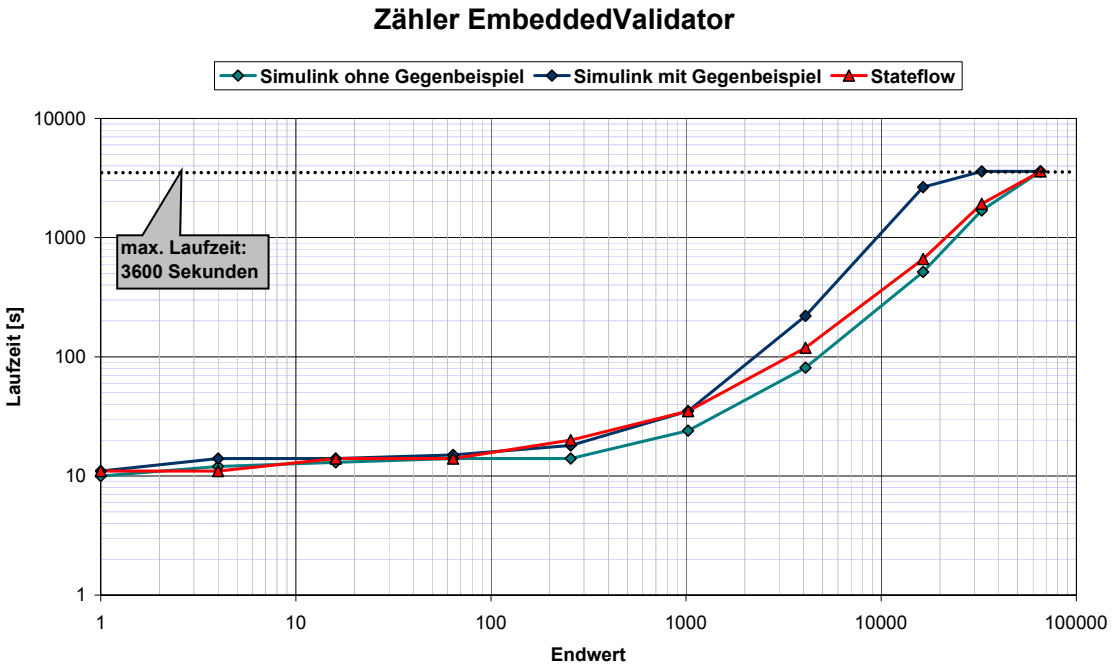


Abbildung 5.1.: Ergebnisse der Effizienzanalyse (Zähler) mit EmbeddedValidator

3. Die Laufzeit für Beweis und Darstellung des Gegenbeispiels wächst schneller als die für den Beweis.

Die Anforderungen für das letzte Submodul der Effizienzanalyse lassen sich mit dem Pattern `init_P_invariant` formulieren. In `P` formulieren wir in einer Disjunktion, dass einer der Eingangswerte gleich dem Ausgangswert ist. Abbildung 5.2 und Tabellen 5.6 zeigen das Verhalten der Laufzeit in Abhängigkeit von der Anzahl und dem Datentyp der Eingänge. Wir erkennen, dass bei bis zu zwei Eingängen der Datentyp für die Laufzeit keine Rolle spielt. Die Laufzeit des Beweises überschreitet bei uint32 schon bei drei Eingängen die Grenze von einer Stunde und bei uint16 geschieht dies bei vier Eingängen. Vergleichende quantitative Aussagen über das Wachstum der Laufzeit mit der Anzahl der Eingänge sind deswegen nicht möglich. Eine höher angesetzte Grenze würde Abhilfe schaffen. Wúnschenswert wären Aussagen für die Modelle mit drei und vier Eingängen.

5.2.2. TNI Safety-Checker Blockset

Wegen der späten Verfügbarkeit von Safety-Checker Blockset überspringen wir bei der Erprobung die Syntaxanalyse. Wir gehen im Anschluss kurz auf die Besonderheiten von Safety-Checker Blockset ein, die sich aus unseren Erfahrungen in der Effizienzanalyse für die Durchführung der Syntaxanalyse ergeben.

5.2.2.1. Effizienzanalyse

Die Anforderung für die Kommunikationsmodelle können wir wie bei EmbeddedValidator nicht vollständig formulieren, AGAF lässt sich nicht ausdrücken. Die Verifikation des ersten Teils der

(a) Simulink

Modell	Laufzeit [s]	
	nur Beweis	Beweis mit Gegenbsp.
SLCounter_0_1_1.mdl	10	11
SLCounter_0_1_4.mdl	12	14
SLCounter_0_1_16.mdl	13	14
SLCounter_0_1_64.mdl	14	15
SLCounter_0_1_256.mdl	14	18
SLCounter_0_1_1024.mdl	24	35
SLCounter_0_1_4096.mdl	81	220
SLCounter_0_1_16384.mdl	515	2650
SLCounter_0_1_32768.mdl	1639	3600
SLCounter_0_1_65536.mdl	3600	3600
SLCounter_0_1_262144.mdl	3600	
SLCounter_0_1_1048576.mdl		
SLCounter_0_1_4194304.mdl		
SLCounter_0_1_16777216.mdl		
SLCounter_0_1_67108864.mdl		
SLCounter_0_1_268435456.mdl		
SLCounter_0_1_1073741824.mdl		
SLCounter_0_1_4294967296.mdl		

(b) Stateflow

Modell	Laufzeit [s]
SFCounter_0_1_1.mdl	11
SFCounter_0_1_4.mdl	11
SFCounter_0_1_16.mdl	14
SFCounter_0_1_64.mdl	14
SFCounter_0_1_256.mdl	20
SFCounter_0_1_1024.mdl	35
SFCounter_0_1_4096.mdl	119
SFCounter_0_1_16384.mdl	661
SFCounter_0_1_32768.mdl	1924
SFCounter_0_1_65536.mdl	3600
SFCounter_0_1_262144.mdl	
SFCounter_0_1_1048576.mdl	
SFCounter_0_1_4194304.mdl	
SFCounter_0_1_16777216.mdl	
SFCounter_0_1_67108864.mdl	
SFCounter_0_1_268435456.mdl	
SFCounter_0_1_1073741824.mdl	
SFCounter_0_1_4294967296.mdl	

Tabelle 5.5.: Ergebnisse der Effizienzanalyse (Zähler) mit EmbeddedValidator

(a) uint8

Modell	Laufzeit [s]
input_1_uint8.mdl	11
input_2_uint8.mdl	14
input_3_uint8.mdl	11
input_4_uint8.mdl	23
input_5_uint8.mdl	657
input_6_uint8.mdl	3600
input_7_uint8.mdl	
input_8_uint8.mdl	3600

(b) uint16

Modell	Laufzeit [s]
input_1_uint16.mdl	11
input_2_uint16.mdl	11
input_3_uint16.mdl	1277
input_4_uint16.mdl	3600
input_5_uint16.mdl	
input_6_uint16.mdl	

(c) uint32

Modell	Laufzeit [s]
input_1_uint32.mdl	10
input_2_uint32.mdl	11
input_3_uint32.mdl	3600
input_4_uint32.mdl	

Tabelle 5.6.: Ergebnisse der Effizienzanalyse (Ein-/Ausgabe) mit EmbeddedValidator

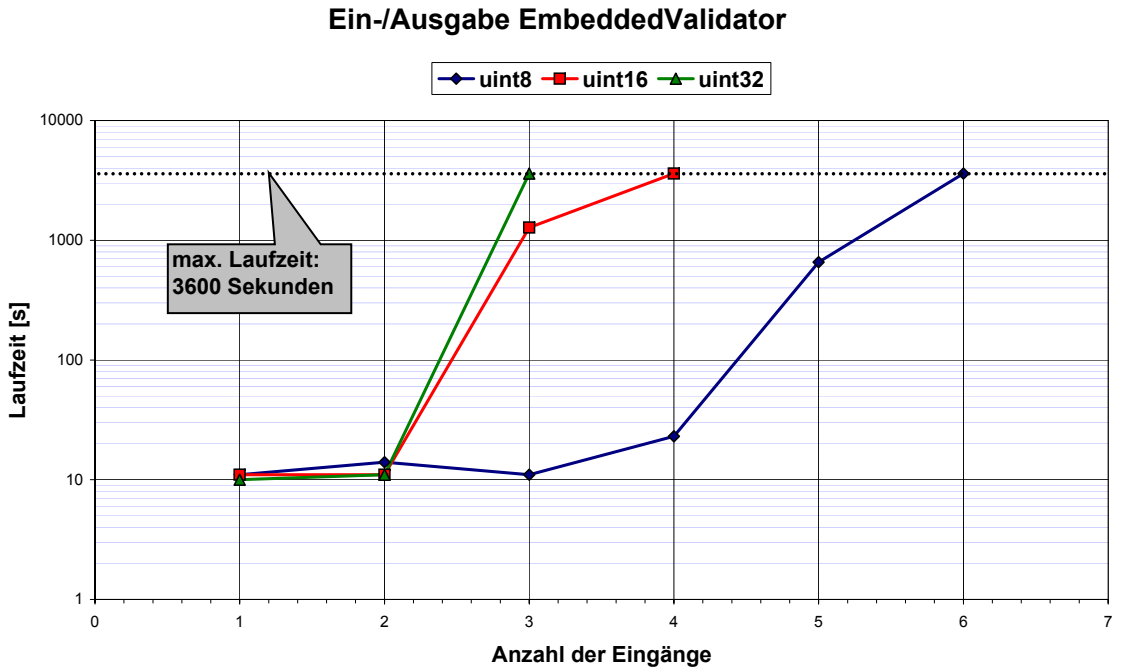


Abbildung 5.2.: Ergebnisse der Effizienzanalyse (Ein-/Ausgabe) mit EmbeddedValidator

Modell	Laufzeit [s]		Bemerkungen
	nur Beweis	Beweis mit Gegenbsp.	
SLCounter_0_1_1.mdl	1	2	
SLCounter_0_1_4.mdl	1	2	
SLCounter_0_1_16.mdl	2	3	
SLCounter_0_1_64.mdl	3	28	
SLCounter_0_1_128.mdl	4	96	
SLCounter_0_1_256.mdl	5	365	
SLCounter_0_1_512.mdl	6	1437	
SLCounter_0_1_1024.mdl	12	3600	
SLCounter_0_1_2048.mdl	22	3600	
SLCounter_0_1_2560.mdl	7		max. Suchtiefe
SLCounter_0_1_3072.mdl	8		max. Suchtiefe
SLCounter_0_1_4096.mdl	8		max. Suchtiefe
SLCounter_0_1_16384.mdl	8		max. Suchtiefe
SLCounter_0_1_65536.mdl			
SLCounter_0_1_262144.mdl			
SLCounter_0_1_1048576.mdl			
SLCounter_0_1_4194304.mdl			
SLCounter_0_1_16777216.mdl			
SLCounter_0_1_67108864.mdl			
SLCounter_0_1_268435456.mdl			
SLCounter_0_1_4294967296.mdl			
SLCounter_0_1_10733741824.mdl			

Tabelle 5.7.: Ergebnisse der Effizienzanalyse (Zähler, nur Simulink) mit Safety-Checker Blockset

Anforderung schlägt fehl. Safety-Checker Blockset überprüft die Anforderung nicht nur in jedem Simulink-Teilschritt, sondern auch zwischen Aktivitäten in einem Stateflow-Chart, die innerhalb eines einzigen Zeitschritts passieren. Parallele Charts werden wie bei der Simulation sequenziell innerhalb eines Zeitschritts abgearbeitet. Dabei treten Konfigurationen der Stateflow-Charts auf, bei denen sich die Prozessor-Automaten nicht in den gleichen Zuständen befinden, obwohl die Anforderung am Ende des Zeitschritts erfüllt ist. Somit können wir keine Effizienzanalyse mit den Kommunikationsmodellen durchführen.

Bei der Effizienzanalyse mit den Simulink-Zählern messen wir zwei Zeiten (Tabelle 5.7). Die erste Zeit ist die Zeitspanne bis eine Aussage vorliegt, ob die Anforderung erfüllt ist. Die zweite beinhaltet zusätzlich die Dauer bis zur Ausgabe eines Gegenbeispiels. Wie schon bei der Erprobung mit EmbeddedValidator formulieren wir die Anforderung negiert (AR statt EU). Bereits bei einer Zählergrenze zwischen 2048 und 2560 erreicht Safety-Checker Blockset seine maximale Suchtiefe und bricht den Beweis ab. Die Stateflow-Varianten des Zählers wurden nicht akzeptiert. Die Ursache dafür ist aufgrund der Fehlermeldung „SYNTAX_ERROR“ unbekannt. In Abbildung 5.3 werden die Laufzeiten von EmbeddedValidator und Safety-Checker Blockset auf den Simulink-Zählermodellen verglichen. Es fällt positiv für Safety-Checker Blockset auf, dass es für den korrekten Beweis selbst deutlich weniger Zeit als EmbeddedValidator braucht. Auf der negativen Seite stehen

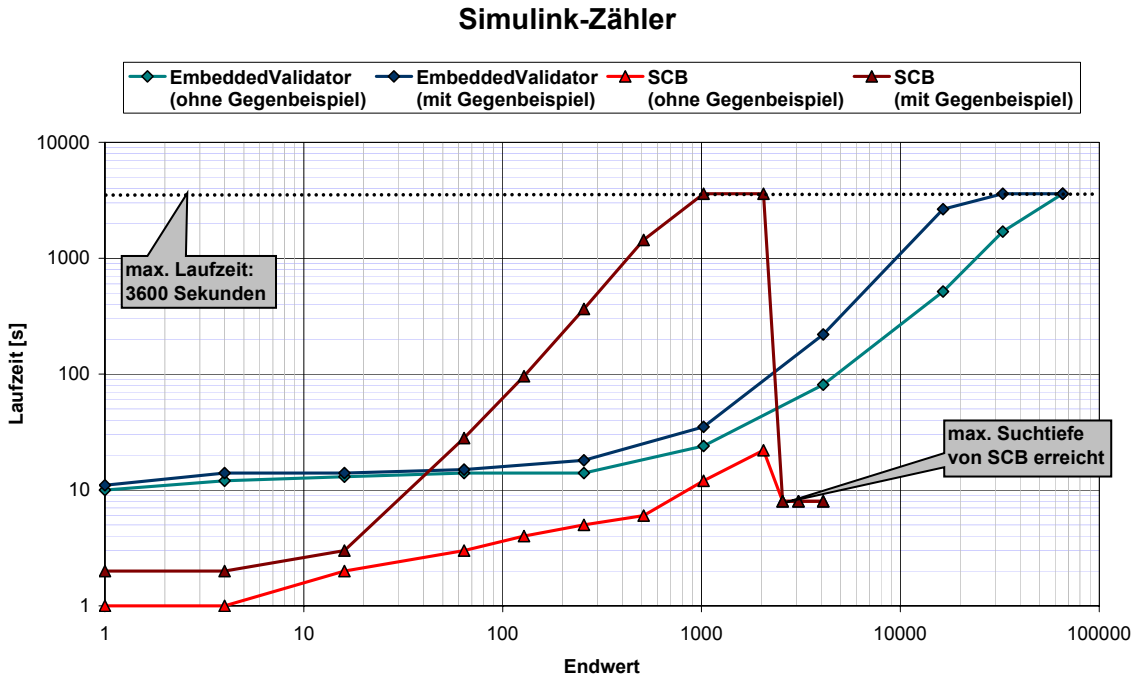


Abbildung 5.3.: Vergleich der Ergebnisse der Effizienzanalyse (Zähler, nur Simulink) mit EmbeddedValidator und mit Safety-Checker Blockset

1. die sehr stark ansteigende Laufzeit bei Erzeugung eines Gegenbeispiels und
2. der stark beschränkte Zustandsraum.

Die Modelle des Ein-/Ausgabe-Submoduls akzeptiert Safety-Checker Blockset. Damit Safety-Checker Blockset die Anforderung korrekt verifiziert, müssen wir jeden Eingang auf einen Wertebereich zwischen 0 und 40 beschränken. Wird einer der Bereiche darüber hinaus erweitert, schlägt der Beweis fehl. Wir können keine Effizienzanalyse durchführen, die mit Ergebnissen anderer Model-Checker vergleichbar wäre. Zum einen entspricht eine Analyse unter solchen Annahmen nicht dem Vorgehensmodell. Zum anderen wäre man gezwungen, bereits früher durchgeführte Analysen erneut durchzuführen, wenn ein anderer Model-Checker ähnliche Beschränkungen aufweist.

5.2.2.2. Anmerkungen zur Syntaxanalyse

Zu erwartende Schwierigkeiten haben wir in 2.3.2.2 behandelt. Daraus und aus der Effizienzanalyse ergeben sich folgende Überlegungnen zur Syntaxanalyse mit Safety-Checker Blcokset.

Bei einer Syntaxanalyse müssen wir beachten, dass das Ergebnis eines einzelnen Tests nicht nur von der Akzeptanz eines Modells abhängt. Ergibt die Überprüfung der korrekten Verarbeitung eines Modells einen Fehlschlag, müssen unter Umständen die Eingangsignale beschränkt werden. Auch kann es sein, dass ein Block abstrahiert wird. Liegt ein solches Verhalten vor, äußert es sich in einem Fehlschlag im ersten Zeitschritt, das dazugehörige Gegenbeispiel ist dabei nicht korrekt.

Safety-Checker Blockset verlangt, dass alle Eingänge eines Blocks vom gleichen Datentyp sind. Wir erwarten deshalb Fehlschläge bei den Modellen des Datentypteils, bei denen zwei Konstanten verschieden Datentyps verglichen werden.

6. Fazit und Ausblick

Zu Beginn lernten wir die Grundlagen des Model-Checking und von Matlab/Simulink kennen und haben daraufhin funktionale und nicht-funktionale Anforderungen an Model-Checker für Simulink-Modelle formuliert. Es stellte sich die Frage, wie wir in kurzer Arbeitszeit objektiv überprüfen können, ob ein beliebiger Model-Checker für Simulink diese Anforderungen erfüllt.

Mit der Evaluierungssuite wurde ein Werkzeug für diese Aufgabe entworfen und implementiert. Es handelt sich dabei um einen Satz von einfachen Simulink-Modellen, mit dem wir bewerten können, ob ein Model-Checker einen Großteil der funktionalen Anforderungen erfüllt. Der Schwerpunkt liegt dabei auf Tests der syntaktischen Fähigkeiten des Model-Checkers bei der Eingabe eines Simulink-Modells.

Bei der Formulierung der Anforderungen haben wir festgestellt, dass wir keine konkreten Anforderungen bezüglich der Effizienz stellen können. Uns fehlte ein Messwerkzeug, mit dem wir die Laufzeit von Model-Checkern in Relation zu Modellart und -größe beurteilen können. Daraus ergab sich eine weitere Aufgabe der Suite. Dafür wurden drei weitere Gruppen von Modellen erstellt, die jeweils in der Zustandsraumgröße variieren. Die Gruppen unterscheiden sich in den syntaktische Merkmalen und den Spezifikationen, gegen die sie verifiziert werden sollen. Mit diesen Modell-Sätzen messen wir die Laufzeit eines Model-Checkers bei unterschiedlichten Zustandsraumgrößen.

Das Vorgehensmodell hilft, die Evaluierung schnell und mit wenigen Tests durchzuführen. Es legt eine Reihenfolge fest, in der die Modelle dem Model-Checker eingegeben werden. Je nach nachdem, ob der Model Checker ein Modell korrekt akzeptiert oder nicht, ergeben sich weitere Tests, oder Schlussfolgerungen, die weitere Test überflüssig machen.

Dass die Evaluierungssuite ihren Zweck gut erfüllt, haben wir überprüft, indem wir zwei Model-Checker für Simulink mit der Suite evaluiert haben. Durch diese Erprobung haben wir sowohl Erkenntnisse über die Evaluierungssuite als auch über die beiden Model-Checker erlangt.

6.1. Bewertung der Model-Checker

OSC EmbeddedValidator hat einen großen Teil der erforderlichen syntaktischen Anforderungen aus Abschnitt 3.1 erfüllt. Auffällig waren Probleme beim Erkennen mancher Blöcke, deren korrekte Akzeptanz wir als ERFORDERLICH eingestuft haben, wie dem Memory-Block (Abschnitt 5.2.1). Diese Probleme waren zum Teil von TargetLink verursacht, teilweise von EmbeddedValidator selbst.

Die Notwendigkeit von TargetLink zur Codegenerierung erwies sich nicht nur aus diesem Grund als problematisch. Es ergeben sich zusätzliche Kosten, wenn man TargetLink nicht bereits nutzt. Der Arbeitsablauf erweist sich als etwas umständlich, da wir das Simulink-Modell erst in ein TargetLink-System übersetzen mussten, und dieses erst als Eingabe für EmbeddedValidator dient. Mit diesem Übersetzungsschritt kommt zudem eine weitere mögliche Fehlerquelle hinzu. Inwieweit sich der über EmbeddedValidator erzeugte C-Code für den Einsatz auf der Zielplattform eignet, haben wir nicht untersucht.

Aus Zeitmangel erfolgte mit TNI Safety-Checker Blockset nur die Effizienzanalyse. Die zu erwartenden Probleme bei der Syntaxanalyse haben wir bereits in 5.2.2 besprochen.

Die Effizienzanalyse konnte für keinen der beiden Model-Checker komplett durchgeführt werden, da keinem der beiden Model-Checker eine dem CTL-Operator AGAF entsprechende Spezifikation eingeben werden konnte. Bei Safety-Checker Blockset kamen weitere Probleme hinzu, wie fehlerhafte Semantik von Stateflow-Charts und starke Einschränkungen der Zustandsraumgröße. Aus diesem Grund lassen sich vergleichende Aussagen zur Effizienz beider Model Checker schwer treffen.

Das Konzept von Safety-Checker Blockset, der als Simulink-Toolbox implementiert ist, ist sehr reizvoll. Man spart sich Zwischenschritte wie mehrfache Übersetzungen. Allerdings macht Safety-Checker Blockset im Vergleich zu EmbeddedValidator einen unausgereiften Eindruck. Wünschenswert wäre ein Model-Checker, der direktes Model-Checking von Simulink-Modellen wie Safety-Checker Blockset durchführt aber mindestens die syntaktischen Möglichkeiten und die Effizienz von EmbeddedValidator bietet.

6.2. Bewertung der Evaluierungssuite

Mit der Syntaxanalyse der Evaluierungssuite war es uns möglich zu überprüfen, welche Blöcke EmbeddedValidator akzeptiert. Wir haben Unterschiede zwischen einzelnen Ausprägungen eines Blocks ausmachen können, wie zum Beispiel beim Math-Operator-Block. Auch problematische Fälle, wie der Fall des Indexstarts beim Multiport-Switch ist, sind aufgefallen. Es ist möglich, diese Informationen durch das Studium der Dokumentation zu gewinnen. Das könnte sich als sehr langwierig erweisen, da man mit zwei Softwarepaketen und deren Dokumentationen arbeitet.

Im Verlauf der Syntaxanalyse hat sich ergeben, dass eine Vorgabe von Spezifikationen, mit denen wir überprüfen können, ob Blöcke korrekt akzeptiert werden, den Evaluierungsprozess beschleunigen würden. Oft mussten wir uns bei einem Test eine geschickte Spezifikation überlegen. Zudem sind solche Spezifikationen notwendig, wenn wir die Aussagen der Syntaxanalyse von mehreren Model-Checkern vergleichen wollen.

Aus dem Studium der Dokumentation von Safety-Checker Blockset ergab sich, dass diese Toolbox bei vielen Blöcken Ausprägungen mit drei Eingängen korrekt akzeptiert, aber nicht Ausprägungen mit mehr als fünf Eingängen. Da wir bei solchen Blöcken Ausprägungen mit maximal einen Eingang mehr als Standard (oft zwei Eingänge) in die Suite aufgenommen haben, würden uns solche Einschränkungen bei einer Syntaxanalyse nicht auffallen.

Die Effizienzanalyse hat, wenn die Modelle akzeptiert und sich die Spezifikationen formulieren ließen, aussagekräftige Ergebnisse geliefert. Die von uns gewählte Zeitbeschränkung auf eine Stunde pro Beweis kann flexibel gehandhabt werden. Steht mehr Zeit zur Verfügung, kann man mehr Messwerte sammeln.

Beim Entwurf der Evaluierungssuite haben wir uns explizit auf die syntaktischen Fähigkeiten bei der Akzeptanz von Modellen und auf die Effizienzanalyse beschränkt. Außen vor haben wir die Untersuchung der Spezifikationseingabesprache des Model-Checkers gelassen. Eine systematische Untersuchung der Möglichkeiten eines Model-Checkers würde folgende Schritte erfordern:

1. Sammlung von typischen Anforderungen an eingebettete Systeme,
2. Extraktion besonderer Merkmale dieser Anforderungen,
3. Formulierung von Spezifikationen, die diese Aspekte betonen,

4. Erstellung möglichst weniger Suite-Modelle, bei denen der Model-Checker überprüfen soll, ob sie die Spezifikationen erfüllen.

Bis auf die Effizienzanalyse haben wir mit der Evaluierungssuite keine nicht-funktionalen Anforderungen an Model-Checker systematisch überprüft. Inwiefern sich dieses im Rahmen einer Erweiterung der Evaluierungssuite bewerkstelligen ließe, muss noch untersucht werden. Interessant ist zudem eine Erweiterung der Effizienzanalyse um einen Satz von Modellen, die der Model-Checker Überschreitungen von Wertebereichen untersuchen soll.

Es ist zu erwarten, dass beim weiteren Einsatz der Suite sich weitere Aspekte bezüglich Erweiterungen und möglichen Verbesserungen der Suite ergeben werden. Der Aufbau der Modelle aus einer Bibliothek und das modulare Vorgehensmodell ermöglichen es, diese unkompliziert in die Evaluierungssuite einzubinden.

Literaturverzeichnis

[ABRW04] Angermann, Anne, Michael Beuschel, Martin Rau und Ulrich Wohlfarth: *Matlab – Simulink – Stateflow: Grundlagen, Toolboxen, Beispiele.* Oldenbourg, 3. Auflage, 2004.

[AD94] Alur, Rajeev und David L. Dill: *A theory of timed automata.* Theoretical Computer Science, 126(2):183–235, 1994.

[BHSV+96] Brayton, Robert K., Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy und Tiziano Villa: *VIS: A System for Verification and Synthesis.* In: Alur, Rajeev und Thomas A. Henzinger (Herausgeber): *CAV*, Band 1102 der Reihe *Lecture Notes in Computer Science*, Seiten 428–432. Springer, 1996.

[CGP99] Clarke, Edmund M., Orna Grumberg und Doron A. Peled: *Model Checking.* MIT Press, 1999.

[dSP04] dSPACE GmbH: *TargetLink Production Code Generation Guide*, 2004.

[Hen96] Henzinger, Thomas A.: *The Theory of Hybrid Automata.* In: *LICS*, Seiten 278–292, 1996.

[Mat05a] The MathWorks, Inc.: *Simulink Reference*, 2005. `http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/slref.pdf`.

[Mat05b] The MathWorks, Inc.: *Stateflow and Stateflow Coder User's Guide*, 2005. `http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf`.

[Mat05c] The MathWorks, Inc.: *Using Simulink*, 2005. `http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf`.

[Mat05d] The MathWorks, Inc.: *Writing S-Functions*, 2005. `http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sfunctions.pdf`.

[OSC05] OSC–Embedded Systems AG: *EmbeddedValidator 2.0 User Guide*, 2005.

[Tiw02] Tiwari, A.: *Formal semantics and analysis methods for Simulink Stateflow models.* Technischer Bericht, SRI International, 2002. `http://www.csl.sri.com/users/tiwari/stateflow.html`.

[TL03] Thomas, Wolfgang und Christoph Löding: *Model-Checking*. Vorlesungsfolien, 2003. `http://www-i7.informatik.rwth-aachen.de/d/teaching/ws0304/modelchk/`.

[TNI05] TNI-Software: *Safety-Checker Blockset V2.2 User's Manual*, 2005.

A. Anhang

A.1. Zusätzliche Definitionen

Definition A.1.1 (Kripke-Struktur)**.** Sei $P = \{p_1, \ldots, p_n\}$ eine Menge atomarer Eingeschaften. Eine Kripke-Struktur $\mathfrak{K} = (S, R, \lambda, s)$ über P besteht aus

- einer endlichen Knotenmenge S,
- einer Menge gerichteter Kanten R,
- einer Beschriftungsfunktion $\lambda : S \to 2^P$ und
- einem ausgezeichneten Anfangsknoten $s \in S$

Definition A.1.2 (Abwicklungsbaum)**.** Sei $\mathfrak{K} = (S, R, \lambda, s)$ eine Kripke-Struktur über P. Ihr Abwicklungsbaum ist eine Kripke-Struktur $\mathfrak{K}_\mathfrak{T} = (S_\mathfrak{T}, R_\mathfrak{T}, \lambda_\mathfrak{T}, s_\mathfrak{T})$ über P mit

- $S_\mathfrak{T} = \{s_0 s_1 \ldots s_n : s_0 = s \wedge (s_i, s_{i+1}) \in R \text{ für alle } 0 \leq i < n\}$,
- $R_\mathfrak{T} = \{(s_0 \ldots s_n, s_0 \ldots s_n s_{n+1}) : (s_n, s_{n+1}) \in R\}$,
- $\lambda_\mathfrak{T}(s_0 \ldots s_n) = \lambda(s_n)$ und
- $s_\mathfrak{T} = s$

Definition A.1.3 (Syntax von CTL*)**.** Sei $P = \{p_1, \ldots, p_k\}$ CTL* ist induktiv aus **Zustandsformeln** und **Pfadformeln** aufgebaut:

Zustandsformeln
- Ist $p \in P$, ist p eine Zustandsformel.
- Sind f_1 und f_2 Zustandsformeln, dann auch $\neg f_1$ und $f_1 \wedge f_2$.
- Ist g eine Pfadformel, ist $\mathsf{E}g$ eine Zustandsformel.

Pfadformeln
- Ist f eine Zustandsformel, ist f auch eine Pfadformel.
- Sind g_1 und g_2 Pfadformeln, dann auch $\neg g_1$, $g_1 \wedge g_2$, $\mathsf{X}g_1$ und $g_1 \mathsf{U} g_2$.

Definition A.1.4 (Semantik von CTL*)**.** Seien f_1, f_2 Zustandformeln, g_1, g_2 Pfadformeln, $\mathfrak{K}_\mathfrak{T}$ der Abwicklungsbaum einer Kripke-Struktur, s ein Knoten in $\mathfrak{K}_\mathfrak{T}$, π ein Pfad in $\mathfrak{K}_\mathfrak{T}$. Die Modellrelation $\models$ ist induktiv definiert.

$$
\begin{array}{ll}
(\mathfrak{K}_\mathfrak{T}, s) \models p_i & :\Leftrightarrow \lambda_{\mathfrak{K}_\mathfrak{T}}(s) = p_i \\
(\mathfrak{K}_\mathfrak{T}, s) \models f_1 \wedge f_2 & :\Leftrightarrow (\mathfrak{K}_\mathfrak{T}, s) \models f_1 \text{ und } (\mathfrak{K}_\mathfrak{T}, s) \models f_2 \\
(\mathfrak{K}_\mathfrak{T}, s) \models \neg f_1 & :\Leftrightarrow (\mathfrak{K}_\mathfrak{T}, s) \not\models f_2 \\
(\mathfrak{K}_\mathfrak{T}, s) \models \mathsf{E}g_1 & :\Leftrightarrow \text{es existiert ein Pfad } \pi, \text{ der in } s \text{ beginnt, mit } (\mathfrak{K}_\mathfrak{T}, \pi) \models g_1 \\
(\mathfrak{K}_\mathfrak{T}, \pi) \models f_1 & :\Leftrightarrow \text{s ist der erste Zustand in } \pi \mathit{und} (\mathfrak{K}_\mathfrak{T}, s) \models f_1 \\
(\mathfrak{K}_\mathfrak{T}, \pi) \models g_1 \wedge g_2 & :\Leftrightarrow (\mathfrak{K}_\mathfrak{T}, \pi) \models g_1 \text{ und } (\mathfrak{K}_\mathfrak{T}, \pi) \models g_2 \\
(\mathfrak{K}_\mathfrak{T}, \pi) \models \neg g_1 & :\Leftrightarrow (\mathfrak{K}_\mathfrak{T}, \pi) \not\models g_1 \\
(\mathfrak{K}_\mathfrak{T}, \pi) \models \mathsf{X}g_1 & :\Leftrightarrow \left(\mathfrak{K}_\mathfrak{T}, \pi^1\right) \models g_1 \\
(\mathfrak{K}_\mathfrak{T}, \pi) \models f_1 \mathsf{U} f_2 & :\Leftrightarrow \text{ es existiert ein } k \geq 0, \text{ so dass } \left(\mathfrak{K}_\mathfrak{T}, \pi^k\right) \models g_2 \text{ und} \\
 & \qquad \text{für alle } 0 \leq i < k \left(\mathfrak{K}_\mathfrak{T}, \pi^i\right) \models g_1
\end{array}
$$

Bemerkung A.1.5. Abkürzend werden oft noch folgende Operatoren benutzt:

$$\begin{aligned} \mathsf{A}g &::= \neg\mathsf{E}\neg g &&: \text{„auf allen ausgehenden Pfaden gilt } g\text{“} \\ \mathsf{F}f &::= \text{true}\mathsf{U}f &&: \text{„auf dem Pfad gilt irgendwann } f\text{“} \\ \mathsf{G}f &::= \neg\mathsf{F}\neg f &&: \text{„auf dem Pfad gilt immer } f\text{“} \\ f_1\mathsf{R}f_2 &::= \neg\left(\neg f_1\mathsf{U}\neg f_2\right) &&: \text{„solange } g_1 \text{nicht gilt, gilt } g_2\text{“} \end{aligned}$$

Definition A.1.6 (Syntax von CTL)**.** CTL ist eine Teilmenge von CTL*, bei der die Pfadformeln eingeschränkt ist.

Zustandsformeln sind wie bei CTL* definiert.

Pfadformeln
- Sind g_1 und g_2 Zustandsformeln, sind $\neg g_1$, $g_1 \wedge g_2$, $\mathsf{X}g_1$ und $g_1\mathsf{U}g_2$ Pfadformeln.

Bemerkung A.1.7. Aus der Definition ergeben sich zehn CTL-Operatoren: AX, EX, AF, EF, AG, EG, AU, EU, AR und ER.

Definition A.1.8 (Syntax von LTL)**.** LTL ist eine Teilmenge von CTL*, die nur aus Pfadformeln aufgebaut ist.

Pfadformeln
- Ist $p \in P$, ist p eine Pfadformel.
- Sind g_1 und g_2 Pfadformeln, dann auch $\neg g_1$, $g_1 \wedge g_2$, $\mathsf{X}g_1$ und $g_1\mathsf{U}g_2$.

Bemerkung A.1.9. Die Semantiken von CTL und LTL ergeben sich aus der Semantik von CTL*, wenn man die entsprechenden syntaktischen Einschränkungen anwendet.

A.2. Semantik von Stateflow

Die Semantik bei der Ausführung von Stateflow wird in Kapitel 3 von [Mat05b] beschrieben. Dabei handelt es sich nicht um eine formale Definition, sondern es wird informell erläutert, wie Stateflow-Charts abgearbeitet werden.

Folgende axiomatische Eigenschaften bilden den Rahmen für das Verhalten von Stateflow.

1. *Ist ein Zustand aktiv, sind seine Oberzustände aktiv.*
2. *Ein Zustand oder Chart mit exklusiver Aufteilung hat maximal einen aktiven Unterzustand.*
3. *Ist ein paralleler Zustand aktiv, sind seine Nebenzustände mit höherer Priorität aktiv.*

Ferner gelten folgende Regeln, damit die Zustände konsistent sind:

1. *Ein aktiver OR-Zustand mit mindestens einem Unterzustand hat genau einen aktiven Unterzustand.*
2. *Alle Unterzustände eines aktiven parallelen Zustands sind aktiv.*
3. *Alle Unterzustände eines inaktiven Zustands sind inaktiv.*

A.2.1. Ausführen eines Ereignisses

Ein Stateflow-Chart läuft genau dann ab, wenn ein Ereignis aufgetreten ist. Das geschieht auf zwei Ebenen:

1. Simulink aktualisiert das Chart, wodurch dieses *aufgeweckt* wird.
2. Ist das Chart wach, reagiert es auf weitere Ereignisse. Tritt kein weiteres Ereignis auf, *ruht* das Chart, bis es von einem Ereignis erneut geweckt wird.

Stateflow läuft in einem Thread ab, so daß alle Aktionen, die aufgrund eines Ereignis erfolgen, für dieses Ereignis atomar erfolgen. Diese Aktionen werden komplett durchgeführt, bevor zu dem dem Ereignis vorgehenden Verhalten zurückgekehrt wird. Die Ausführung kann nur durch ein *early return* unterbrochen werden (s. A.2.5).

Es gibt zwei Quellen für Stateflow Ereignisse.

Simulink: Charts werden von Simulink-Ereignissen aufgeweckt. Ist ein Chart aktiv, werden weitere Ereignisse von aussen wie normale Ereignisse behandelt.

Gesendete Ereignisse (*broadcast events*): Durch diese Ereignisse kann der Ablauf des Charts gesteuert werden. Sie werden in dem Chart mit der *action language* definiert.

Ereignisse haben einen Besitzer (*parent*) und einen Gültigkeitsbereich (*scope*). Diese definieren den Zugriff auf das jeweilige Ereignis.

Tritt ein Ereignis auf, wird es durch von der obersten Hierarchieebene (Wurzel) des Charts aus nach unten durchgereicht. Auf jeder Ebene werden bei den aktiven Zuständen during- und passende on *event_name*-Aktionen durchgeführt und die Unterzustände auf passende Transitionen untersucht.

Alle Ereignisse werden in Stateflow wie folgt behandelt:

1. Ist der *Empfänger* des Ereignisses aktiv, wird er ausgeführt. Er ist der Besitzer des Ereignisses, ausser das Ereignis wurde ihm explizit durch einen send()-Befehl gesandt.
2. Ist der Empfänger nicht aktiv, passiert nichts.
3. Der Sender des Ereignisses führt *early return logic* aus, abhängig vom Ereignis, welches das Senden des Ereignisses bedingt hat (s. A.2.5).

A.2.2. Ausführen eines Charts

Ein Chart läuft ab, wenn es von einem Simulink-Ereignis angestoßen wird. Das Ereignis wird von da an wie ein gewöhnliches Ereignis behandelt. Enthält ein Chart keine aktiven Zustände, ist es inaktiv, sonst entweder ruhend oder aktiv.

Wird ein inaktives Chart von einem Ereignis angestoßen, werden erst die Standardflussgraphen behandelt. Wird dabei kein Zustand betreten und das Chart ist parallel aufgeteilt, wird das Ereignis an jeden Unterzustand weitergegeben, und diese betreten. Wird kein Zustand betreten, so sind die Zustände inkonsistent und das Chart wird im weiteren Verlauf der Simulation nicht ausgeführt.

Empfängt ein aktives Chart ein Ereignis, werden seine aktiven Unterzustände ausgeführt.

Abbildung A.1 zeigt den Standardablauf nach Auftreten eines Simulink-Ereignisses. Early-Return und Event-Broadcast wurden der Einfachheit halber nicht aufgenommen.

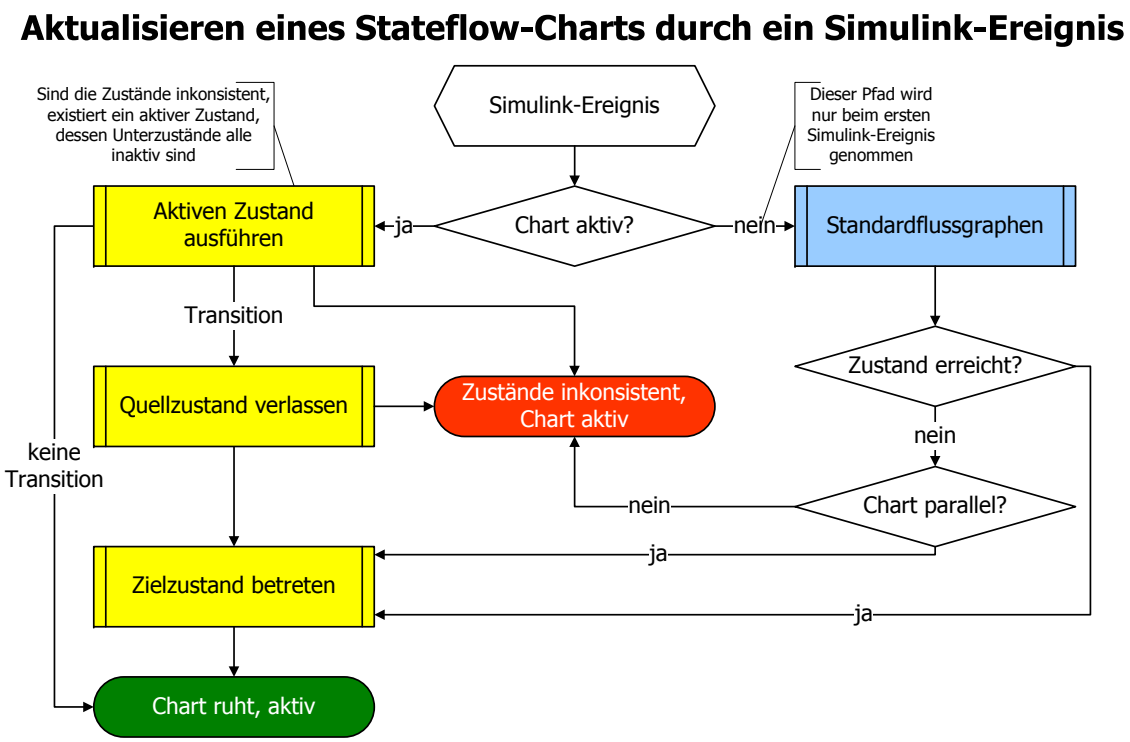


Abbildung A.1.: Schritte beim Auftreten eines äußeren Ereignisses

A.2.3. Ausführen einer Transition

Bevor Transitionen für einen aktiven Zustand ausgeführt werden, werden sie in drei Mengen von Flussgraphen zusammengefasst.

- Alle Transitionssegmente, die Stardardtransitionssegmente enthalten und im gleichen Oberzustand starten, gehören zu Standardflussgraphen.
- Innere Flussgraphen umfassen alle Segmente, die in einem Zustand starten und diesen nicht verlassen.
- Äußere Flussgraphen enthalten alle Segmente, die den Quellzustand verlassen.

Diese Mengen müssen nicht disjunkt sein. Die obige Reihenfolge legt die Prioritäten unter den Mengen fest.

Folgende Schritte werden für jede der Mengen durchgeführt.

1. Die ausgehenden Transitionen des aktiven Zustands werden geordnet (s. Abschnitt A.2.3.1).
2. Das nächste Segment wird ausgewählt.
3. Das gewählte Segment wird auf seine Gültigkeit geprüft.
4. Ist das Segment nicht gültig, wird zu Schritt 2 gesprungen.
5. Ist das Ziel des Segments ein Knotenpunkt mit ausgehenden Transitionssegmenten, werden die Schritte ab 1 für diese Segmente durchgeführt.
6. Ist das Ziel des Segments ein Knotenpunkt ohne ausgehende Transitionssegmente, endet das Testen, ohne daß ein Zustand betreten oder verlassen wurde.
7. Ist das Ziel des Segments ein Zustand, dann:
   a) Es werden keine weiteren Segmente getestet und ein Transitionspfad aus allen bisher besuchten, in Kreuzungen beginnenden Segmenten wird gebildet.
   b) Die unmittelbaren Unterzustände des Quellzustands der Transition werden verlassen.
   c) Die Transitionsaktion des letzten Segments des Transitionspfades wird ausgeführt.
   d) Der Zielzustand wird betreten.
8. Wurden alle Segmente einer Flussgraphenmenge einer Kreuzung überprüft, wird das Segment, mit der der Knotenpunkt erreicht wurde, zurückgegangen und dort mit Schritt 2 fortgefahren.

Abbildung A.2 bietet eine Übersicht, wie eine Flussgraphenmenge abgearbeitet wird.

A.2.3.1. Ordnung der Transitionen

Verläßt mehr als ein Transitionssegment eine Quelle (Zustand oder Knotenpunkt), muss eine Reihenfolge der Überprüfung festgelegt werden. Diese kann der Benutzer selber festlegen, oder die implizite Ordnung nutzen. Stateflow ermittlet diese mittels dreier Kriterien mit folgender Priorität.

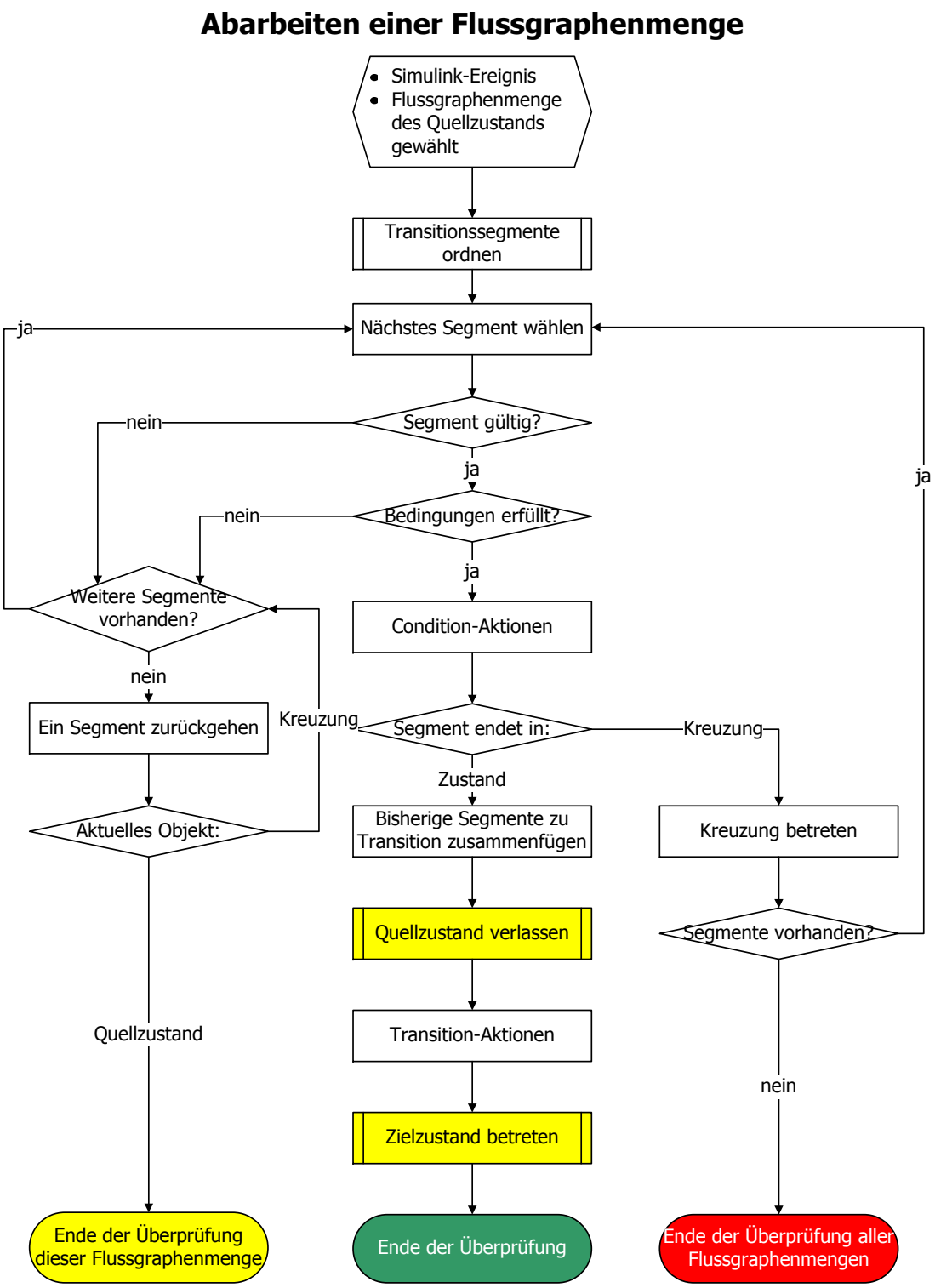


Abbildung A.2.: Abarbeiten einer Flussgraphenmenge

Zielhierarchieebene: Je höher die Hierarchieebene des Ziels liegt, desto früher wird das Segment getestet.

Beschriftung: Segmente mit gleicher Zielhierarchieebene werden nach ihrer Beschriftung geordnet:

1. Beschriftung mit Ereignissen und Bedingungen
2. Beschriftung mit Ereignissen
3. Beschriftung mit Bedinungen
4. Keine Beschriftung

Startposition: Nach Zielhierarchieebene und Beschriftung äquivalente Segmente werden nach der geometrischen Position ihrer Startpunkte geordnet. Dabei werden sie **aufsteigend** im mathematisch positiven Sinn geordnet. Die Transition mit der niedrigsten Priorität liegt bei Zuständen direkt unterhalb der oberen linken Ecke, bei Knotenpunkten in der 12-Uhr-Position. Transitionen von Ecken aus sind nicht erlaubt.

A.2.4. Betreten, Ausführen und Verlassen eines Zustandes

Zustände werden betreten (aktiviert), ausgeführt und anschließend verlassen (als inaktiv markiert).

A.2.4.1. Betreten

Ein Zustand kann auf drei Arten aktiviert werden:

- Eine eingehende Transition tritt über seine Begrenzung.
- Der Zustand ist das Ziel einer Transition.
- Der Zustand ist ein paralleler Unterzustand eines aktivierten Zustands.

Beim Betreten werden folgende Schritte ausgeführt:

1. Ist der Oberzustand nicht aktiv, werden Schritte 1 bis 4 mit diesem durchgeführt.
2. Ist der aktuelle Zustand ein Parallelzustand, werden höher priorisierte Geschwister auf Aktivität überprüft. Diese werden von oben nach unten und von links nach rechts absteigend geordnet. Ist ein solcher Zustand aktiv, wird bei ihm ab Schritt 1 fortgefahren.
3. Der Zustand wird als aktiv markiert.
4. *Entry Actions* werden ausgeführt.
5. Die Unterzustände werden betereten. Bei Bedarf:
   a) Enthält der Zustand keine *History Junction*, werden seine Standardflussgraphen ausgeführt.
   b) Enthält der Zustand eine *History Junction* und ist ein Unterzustand dieses Zustands aktiv, werden die *Entry Actions* für diesen ausgeführt.
   c) Enthält der Zustand parallele Zustände, werden diese der Reihe nach betreten.

6. Ist der Zustand ein Parallelzustand, werden die Eintrittsaktionen bei dem in der Reihenfolge nächsten Zustand durchgeführt.

7. Ist der Empfänger der Transition nicht der Oberzustand des aktuellen Zustands, werden Schritte 6 und 7 beim unmittelbaren Oberzustand ausgeführt.

8. Das Chart ruht.

Abbildung A.3 bietet eine Übersicht, wie ein Zustand betreten wird.

A.2.4.2. Ausführen

Empfängt ein aktiver Zustand ein Ereignis, daß kein Verlassen verursacht, werden folgende Aktionen durchgeführt.

1. Die Menge der äusseren Flussgraphen wird abgearbeitet. Kann dadurch eine Transition genommen werden, wird der Zustand nicht weiter bearbeitet.

2. During- und on *event_name*-Aktionen werden in der angegebenen Reihenfolge durchgeführt.

3. Die Menge der inneren Flussgraphen wird abgearbeitet. Wird keine Transition verursacht, werden die aktiven Unterzustände behandelt, parallele Zustände werden gemäß ihrer Ordnung behandelt.

Abbildung A.4 zeigt die Schritte beim Ausführen eines Zustands.

A.2.4.3. Verlassen

Ein Zustand kann auf drei Arten deaktiviert werden:

- Eine ausgehende ausgeführte Transition startet von seiner Begrenzung.
- Eine ausgehende ausgeführte Transition kreuzt seine Begrenzung.
- Der Zustand ist ein paralleler Unterzustand eines inaktivierten Oberzustands.[1]

Folgende Schritte werden unternommen, wenn ein Zustand verlassen wird.

1. Ist der Zustand ein Parallelzustand und einer seiner Nebenzustände wurde vor diesem betreten, werden die Nebenzustände in der umgekehrten Reihenfolge des Betretens verlassen.

2. Hat der Zustand aktive Nachfolger, werden diese in der umgekehrten Reihenfolge des Betretens inaktiviert.

3. exit-Aktionen werden abgearbeitet.

4. Der Zustand wird als inaktiv markiert.

Wie ein Zustand verlassen wird, zeigt Abbildung A.5.

[1] In [Mat05b] steht, daß der Oberzustand aktiviert sein müsse, was aber nicht sinnvoll ist, da unter dieser Bedingung der Unterzustand schon aktiviert wurde. Es scheint sich um einen Fehler in der Dokumentation zu handeln.

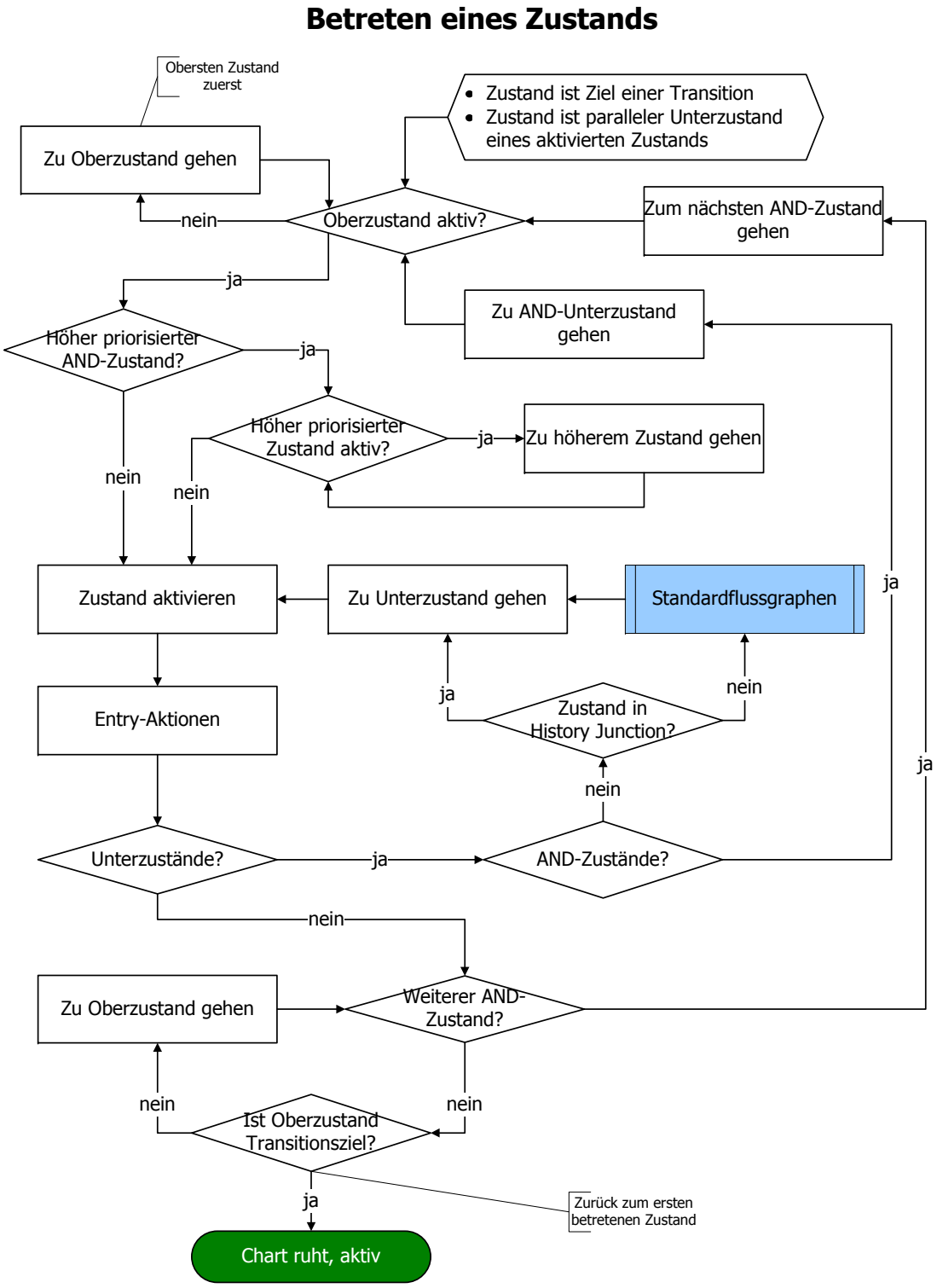


Abbildung A.3.: Betreten eines Zustands

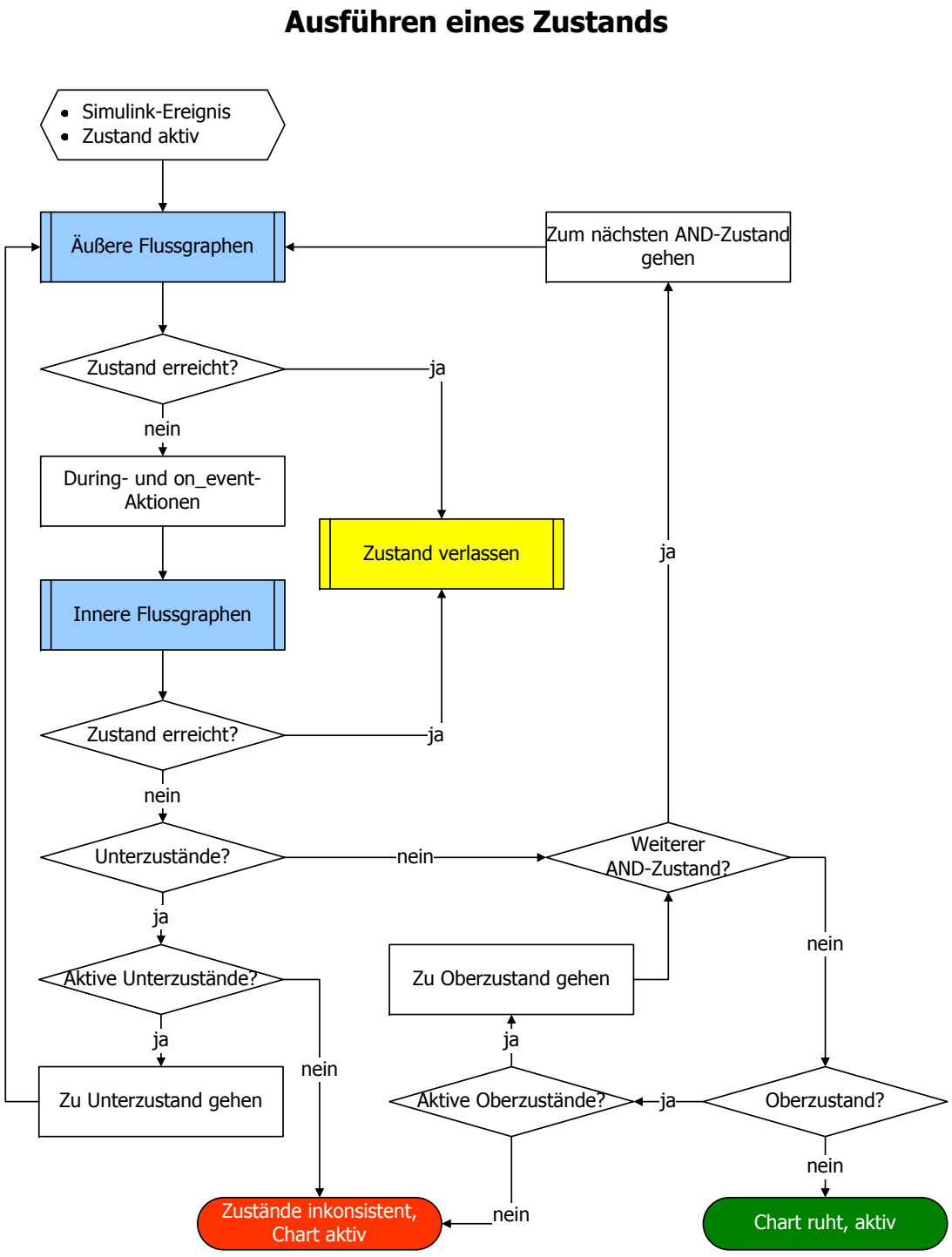


Abbildung A.4.: Ausführen eines Zustands

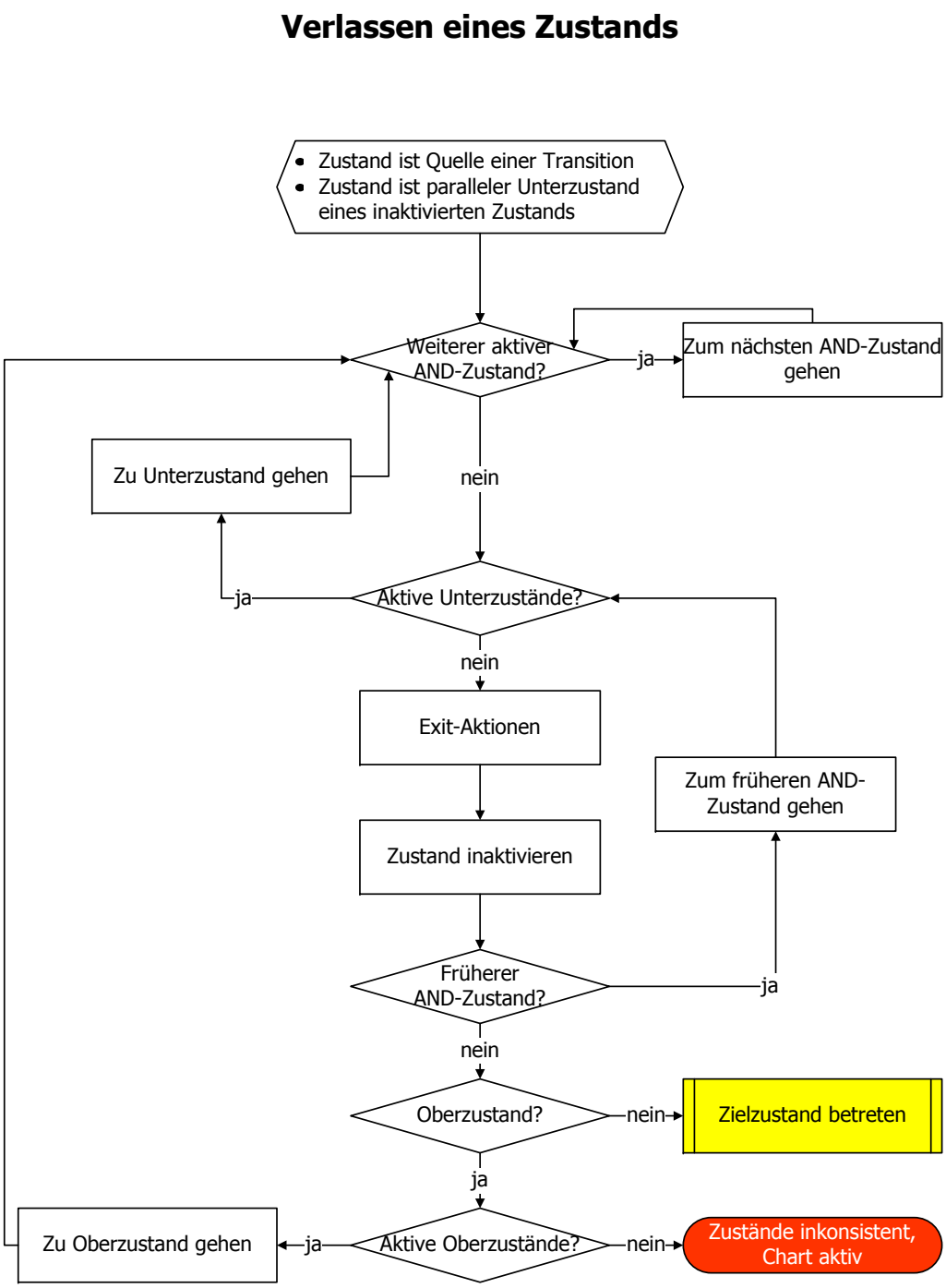


Abbildung A.5.: Verlassen eines Zustands

A.2.5. Verhalten bei *Event Broadcasts*

Stateflow arbeitet in einem einzelnen Thread und muss bei einem Event Broadcast die aktuell ablaufenden Aktionen unterbrechen, um diesen zu bearbeiten. Durch die in A.2 genannten Eigenschaften können Konflikte auftreten, die durch *Early Return Logic* aufgelöst werden. Diese ist vom Typ der den Broadcast verursachenden Aktion abhängig.

Entry Ist der Zustand nach dem Event Broadcast nicht mehr aktiv, wird der Zustand nicht weiter betreten.

During Ist der Zustand nach dem Event Broadcast nicht mehr aktiv, wird der Zustand nicht weiter abgearbeitet.

Exit Ist der Zustand nach dem Event Broadcast nicht mehr aktiv, wird der Zustand nicht weiter verlassen. Insbesondere werden keine Transitionen von Zustand zu Zustand durchgeführt.

Condition Ist der Startzustand des inneren oder äußeren Flussgraphen oder der Oberzustand des Standarsflussgraphen nach dem Event Broadcast nicht mehr aktiv, wird die Menge der Flussgraphen nicht weiter abgearbeitet.

Transition Ist der Oberzustand eines Transitionspfads nach dem Event Broadcast nicht mehr aktiv, oder hat er einen aktiven Unterzustand, werden die verbleibenden Transitionsaktionen nicht durchgeführt und der Zielzustand nicht betreten.

A.3. Schnittstellen zwischen Simulink und Stateflow

Jedes Stateflow-Chart ist ein Block in der Simulink-Umgebung. Daraus ergeben sich diverse Möglichkeiten, wie Stateflow mit seiner Umgebung interagiert.

- Physische Verbindungen zwischen Simulink- und Stateflow-Blöcken sind die übliche Weise, auf die Simulink und Stateflow interagieren.
- Ereignissen und Daten können zwischen Stateflow-Blöcken und externen Quellen ausgetauscht werden.
- Unter den Eigenschaften des Stateflow-Charts ist die Update-Methode für das Verhalten des Charts entscheidend.
- Durch den Export von graphischen Funktionen werden diese anderen Blöcken zur Verfügung gestellt.
- Der Einsatz von Daten und Funktionen vom Matlab-Workspace ist möglich, steht aber nicht mehr zur Verfügung, wenn Code für eine Zielumgebung erzeugt werden soll.
- Definitionen in externen Code-Quellen sind notwendig, wenn auf Ereignisse und Daten von Stateflow-Charts zugegriffen wird.

A.3.1. Ereignisse und Daten

Ereignisse und Daten können zwischen Stateflow- und anderen Simulink-Blöcken über physische Verbindungen ausgetauscht werden. Ferner gibt es die Möglichkeit, externe Quellen und Ziele einzubinden, die aber vorerst aussen vor gelassen wird.

Innerhalb eines Charts sind zusätzlich implizite Ereignisse sichtbar, die bei folgenden Aktivitäten erzeugt werden:

- Chart wird geweckt
- Zustand wird betreten
- Zustand wird verlassen
- Einem internen Datenobjekt wird ein Wert zugewiesen

Diese impliziten Ereignisse werden nur ausgewertet, wenn auf sie mittels Action-Language-Anweisungen zugegriffen wird.

Zahlenwerte werden in Datenobjekten im Stateflow-Chart gespeichert. Der Zeitpunkt, an dem Daten initialisiert werden, hängt vom Oberobjekt und Gültigkeitsbereich des Datums ab:

Initialisierung bei	**Oberobjekt**	**Gültigkeitsbereich**
Simulationsbeginn	Maschine	Local
		Exported
Simulationsbeginn oder Reinitialisierung des Charts	Chart	Output
		Local
	Zustand mit History Junction	Local
	Funktion	Local
Zustandsaktivierung	Zustand ohne History Junction	Local
Funktionsaufruf	Funktion	Input
		Output

Neben Call-by-Value ist für vom Benutzer geschriebenen Funktionen Call-by-Reference möglich.

A.3.1.1. Austausch von Ereignissen

Durch Eingangsereignisse wird ein Chart über bestimmte Vorgänge in der Simulinkumgebung informiert. Stateflow-Blöcke mit Eingangsereignissen erhalten einen Triggereingang mit einer entsprechenden Anzahl Ports. Die eingehenden Ereignisse können den Block auf zwei Arten triggern.
Als *Function-Call*-Ereignis wird der Block wie eine S-Funktion verarbeitet. Alle Eingängsereignisse müssen dann von diesem Typ sein. Weitere Informationen dazu findet man in [Mat05d].
Zum anderen können Ereignisse als Änderungen von Signalen eingehen. Entscheidend dabei ist, ob es in dem Signal einen Nulldurchgang gibt. In welcher Richtung dieser (steigende, fallende oder beide Flanken) für ein Ereignis ablaufen soll, kann angegeben werden. Bei gleichzeitigem Auftreten von mehreren Ereignissen werden diese der Portreihenfolge nach abgearbeitet.

Ereignisse können auch vom Stateflow-Block an andere Simulink-Blöcke übergeben werden. Für jedes Ausgangsereignis wird ein skalarer Ausgang erstellt. Dieser dient als Flanken-Trigger oder als Funktionsaufruf.

A.3.1.2. Austausch von Daten

Wie Ereignisse können auch Daten zwischen dem Stateflow-Block und anderen Blöcken ausgetauscht werden. Für jedes Input-, bzw. Output-Datenobjekt wird ein Ein-, bzw. Ausgang am Stateflow-block geschaffen. Existiert ein Simlink-Signal mit dem Namen eines Ausgangsdatenobjekts des Stateflow-Blocks, wird dieses dem Signal implizit zugeordnet.

Weitere Merkmale des Datenaustauschs:

- Parameter für maskierte Subsysteme können als Daten in einen Stateflow-Block eingebracht werden.
- Neben einer expliziten Typangabe ist es auch möglich, den Stateflow-Block den Eingangsdatentyp vom übergebenden Simulink-Block erben zu lassen.
- Call-by-Reference wird bei Input-Daten nicht empfohlen.

A.3.2. Update von Stateflow-Blöcken

Die Update-Methode bestimmt wann das Stateflow-Chart zur Ausführung aufgeweckt wird. Es stehen drei Methoden zur Auswahl.

Inherited ist die Standardmethode. Eingaben vom Simulink-Modell bestimmen dabei die Weckzeitpunkte. Defniert der Benutzer für den Stateflow-Block Eingangsereignisse, dienen die mit diesen verbundenen Signale als Trigger für den Block (siehe A.3.1). Wurden keine Eingangsereignisse definiert, erbt Stateflow Trigger vom Simulink-Modell. Hat der Block Dateneingänge, wird er mit Rate des schnellsten Dateneingangs aufgeweckt. Andernfalls erbt er das Ausführungsverhalten des ihn umgebenden Simulink-Subsystems.

Discrete: Simulink weckt das Chart mit einer festen Abtastrate. Ein implizites Ereignis wird erzeugt.

Continuous: Simulink weckt das Chart in jedem Simulationsschritt und bei weiteren vom Simulink-Slover verlangten Zeitpunkten.

Die Abtastrate muss bei Wahl eines diskreten Solvers ein ganzzahliges Vielfaches der Solver-Rate sein.

A.4. RFC 2119

```
Network Working Group                                          S. Bradner
Request for Comments: 2119                           Harvard University
BCP: 14                                                        March 1997
Category: Best Current Practice


        Key words for use in RFCs to Indicate Requirement Levels

Status of this Memo

   This document specifies an Internet Best Current Practices for the
```

Internet Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited.

Abstract

In many standards track documents several words are used to signify the requirements in the specification. These words are often capitalized. This document defines these words as they should be interpreted in IETF documents. Authors who follow these guidelines should incorporate this phrase near the beginning of their document:

> The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Note that the force of these words is modified by the requirement level of the document in which they are used.

1. MUST This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.

2. MUST NOT This phrase, or the phrase "SHALL NOT", mean that the definition is an absolute prohibition of the specification.

3. SHOULD This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

4. SHOULD NOT This phrase, or the phrase "NOT RECOMMENDED" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

[...]

5. MAY This word, or the adjective "OPTIONAL", mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

```
6. Guidance in the use of these Imperatives

   Imperatives of the type defined in this memo must be used with care
   and sparingly.  In particular, they MUST only be used where it is
   actually required for interoperation or to limit behavior which has
   potential for causing harm (e.g., limiting retransmisssions)  For
   example, they must not be used to try to impose a particular method
   on implementors where the method is not required for
   interoperability.

7. Security Considerations

   These terms are frequently used to specify behavior with security
   implications.  The effects on security of not implementing a MUST or
   SHOULD, or doing something the specification says MUST NOT or SHOULD
   NOT be done may be very subtle. Document authors should take the time
   to elaborate the security implications of not following
   recommendations or requirements as most implementors will not have
   had the benefit of the experience and discussion that produced the
   specification.

8. Acknowledgments

   The definitions of these terms are an amalgam of definitions taken
   from a number of RFCs.  In addition, suggestions have been
   incorporated from a number of people including Robert Ullmann, Thomas
   Narten, Neal McBurnett, and Robert Elz.
[...]
9. Author's Address

      Scott Bradner
      Harvard University
      1350 Mass. Ave.
      Cambridge, MA 02138

      phone - +1 617 495 3864

      email - sob@harvard.edu
```

Wir haben die Kopfzeilen des ursprünglichen Textes weggelassen.

A.5. Evaluation Suite User Guide

A.5.1. Introduction

The task of this suite is to evaluate different model checking tools for Matlab/Simulink. The evaluation focuses mainly on two aspects:

1. Acceptance of different syntactical constructs (blocks, data types, etc.) is checked in syntax analysis.
2. Efficiency in solving typical proofs is measured in the efficiency analysis.

The models of the suite are distributed in different hierarchically organized directories. For example each aspect is represented by a directory, which contains further subdirectories corresponding to the blocks, data types, etc. characterizing the models.

The main element of each model provided with the evaluation suite is a masked atomic subsystem, which contains the blocks, systems, etc. that have to be verified.

A.5.2. Workflow

There is a defined workflow, to go model by model through the evaluation suite. It provides a way to get all necessary results with as little time need as possible.

You can find the workflow's top-down graphical representation in subdirectory `11_Documentation/01_Workflow/`. The different colors represent the usage of models and if decisions are based on earlier results:

- Gray represents the usage of data gathered during the evaluation.
- Green represents models used in the workflow, if all tests were successful.
- Yellow represents alternative paths in the workflow, taken when results of earlier segments have shown, that a "green model" will not be accepted.
- Orange represents tests that have to be performed after failures in the same segment of the workflow.

Segments are divided by start and end marker.

You start with the syntax analysis and proceed with the efficiency analysis.

A.5.2.1. Preparation

Before using a model, you have to perform some preparations.

1. Copy the evaluation suite in an own work directory. In order to avoid potential confusions caused by too long file and path names, the copy should be near the root directory.
2. Add the directory containing the copy of the evaluation suite and the one containing the library file `lib_eval.mdl` to Matlab's path variable.
3. Disable and break all links in the subsystems to the library.

Depending on the evaluated model checker further preliminary steps may be necessary, for instance:

- Add additional mandatory blocks.
- Unmask Subsystems.

You should know in ahead how you can input the specification.

A.5.2.2. Collecting Results

There are tables prepared to collect the results in a proper way. In `Results.xls` you can enter successful tests, failures and runtimes, respectively. You will find there individual sheets for the basic blocks, data and diagram types, automotive models and the efficiency analysis.

In some cases a result for one model entails results for others. These cases are mentioned in the corresponding rows of the table. Since for some results the conclusions are more complicated, you can find these rules in an own sheet named 'dependency'. Note that in the mentioned cases sometimes the wildcard '*' is used.

A.5.2.3. Syntax Analysis

In this part the user analyzes the model checker's syntactical capabilities. This part is mandatory for the following ones.

The models in the syntax analysis serve as pure acceptance tests. A model is input to the model checker and in order to let the test be successful, the model checker has to recognize it syntactically. Therefore it is recommended that you perform a simple function test.

The syntax analysis consists of three parts:

1. Basic Blocks
2. Data Type
3. Diagram Type

You can find the corresponding workflow in `Basic.pdf`, `DataType.pdf` and `DiagramType.pdf` respectively.

A.5.2.4. Efficiency Analysis

In this part of the evaluation process you evaluate the performance of the model checker. It consists of three parts as you can see in `Efficiency.pdf`:

1. Communication
2. Counter
   a) Simulink
   b) Stateflow
3. IO-Test

As you can see, the incrementing variable 'N' is used in loops in order to calculate the corresponding filename.

To every part of the efficiency analysis belongs a specification, which has to be verified on the models within given time. All specifications are true on the given models. As above the

specification is given here in English and in CTL. You have to adjust the specification to the model's attributes. Further it may be necessary that you have to translate them for the model checker.

Communication The automata synchronize by sending their current state to the following automaton over the media.

In each step all automata are in the same state and every automaton is infinitely often in state 0 and infinitely often in state 1.

$$\mathsf{AG}\left((A_0 = 0 \leftrightarrow \ldots \leftrightarrow A_n = 0) \wedge (A_0 = 1 \leftrightarrow \ldots \leftrightarrow A_n = 1)\right)$$
$$\wedge \bigwedge_{i=0}^{n} \mathsf{AGAF}\,(A_i = 0) \wedge \bigwedge_{i=0}^{n} \mathsf{AGAF}\,(A_i = 1)$$

Counter For both implementations of the counter (Simulink and Stateflow) the same specification should be verified:

There exists a run of the system, such that output "stop" is set to 1 until the output "counter" reaches the end value.

$$\mathsf{E}\,(\mathit{end} = 1\ \mathsf{U}\ \mathit{counter} = \mathit{stop})$$

IO-Test The MinMax block chooses the smallest input as output.

In each step the output value is equal to one of the input values.

$$\mathsf{AG}\left(\bigvee_{i=1}^{n} \mathit{Out} = \mathit{In}_i\right)$$